Privately Outsourcing Exponentiation to a Single Server: Cryptanalysis and Optimal Constructions

Celine Chevalier, Fabien Laguillaumie, Damien Vergnaud

Noname manuscript No.
(will be inserted by the editor)

Privately Outsourcing Exponentiation to a Single Server: Cryptanalysis and Optimal Constructions

Céline Chevalier · Fabien Laguillaumie · Damien Vergnaud*

the date of receipt and acceptance should be inserted later

Abstract We address the problem of speeding up group computations in cryptography using a single untrusted computational resource. We analyze the security of two efficient protocols for securely outsourcing (multi-)exponentiations. We show that the schemes do not achieve the claimed security guarantees and we present practical polynomial-time attacks on the delegation protocols which allow the untrusted helper to recover part (or the whole) of the device's secret inputs. We then provide simple constructions for outsourcing group exponentiations in different settings (*e.g.* public/secret, fixed/variable bases and public/secret exponents). Finally, we prove that our attacks are unavoidable if one wants to use a single untrusted computational resource and to limit the computational cost of the limited device to a constant number of (generic) group operations. In particular, we show that our constructions are actually optimal in terms of operations in the underlying group.

Keywords. Secure outsource computation, Cryptanalysis, Coppersmith methods, Protocols, Optimality results

1 Introduction

We address the problem of "outsourcing" computation from a (relatively) weak computational device to a more powerful entity. This problem has been considered in various settings since many years (distributed-computing projects – *e.g.*, Mersenne prime search – or cloud computing) but the proliferation of mobile devices, such as smart phones or RFID tags, provides yet another venue in which a computationally weak device would like to be able to outsource a costly operation to a third party helper. Low-cost RFID tags do not usually have the computational or memory resources to perform complex cryptographic operations

* Corresponding author: damien.vergnaud@lip6.fr

This is the full and updated version of the ESORICS 2016 paper [13] with additional material.

CRED (U. Panthéon–Assas Paris II) · LIP (UCBL, U. Lyon, CNRS, ENS Lyon, INRIA) · Sorbonne Université, CNRS, LIP6 and Institut Universitaire de France

and it is natural to outsource these operations to some helper. The Near Field Technology (NFC) is embedded in the current generation of cellphone and can be used for transport tickets, credit cards, transit pass, loyalty cards or access control badges. This contactless technology raises many questions of disclosure of sensitive personal information. To preserve privacy, complex anonymity-oriented cryptographic protocols should be used and it is mandatory to delegate some of the costly operations from the chip to the phone, because these protocols are highly resource-consuming. However, in this scenario, this helper (*e.g.* the phone) can, potentially, be operated by a malicious adversary and we usually need to ensure that it does not learn anything about what it is actually computing.

The wild and successful deployment of cloud storage services, like Google Drive, Dropbox, or Amazon Cloud Drive make users outsource their data, for a personal or commercial purpose. These users actually have to trust their storage providers concerning the availability of their data, and indeed outages happen regularly. That is why it has been proposed to audit online storage services [41]. Cryptographic primitives are needed to convince customers (or an external trusted auditor) that their platforms are reliable. Among such primitives, provable data possessions [2] and proofs of retrievability [27] allow the storage cloud to prove that a file uploaded by a client has not been deteriorated or that it can be entirely retrieved. The computation needed on the verification side by the client are highly *"exponentiation-consuming"*.

Indeed, the core operation of these cryptosystems is group exponentiation, i.e., computing u^a from a group element u and an exponent a. The main goal of this paper is to analyze new and existing protocols outsourcing group exponentiation to an untrusted helper.

1.1 Prior work

One can date back the first protocol for securely outsourcing group exponentiation to the precomputation scheme proposed by Schnorr in his seminal paper on discrete-logarithm based signatures [39, Section 4]. Schnorr proposed a scheme for fast generation of pairs (g^k, k) where g is a generator of a cyclic group $\mathbb{G} = \langle g \rangle$ of prime-order p and k is a (purported) random element in $\mathbb{Z}_p$. The scheme was broken by de Rooij for the small parameters suggested by Schnorr (see [19]) but new proposals with provable security were proposed subsequently (see [8] and references therein).

Even if the problem of outsourcing cryptographic operations is not new, it has known a revival of interest in the last ten years with the development of mobile technologies. In 2005, Hohenberger and Lysyanskaya [25] provided a formal security definition for securely outsourcing computations from a computationally limited device to untrusted helpers and they presented two practical schemes. Their first scheme shows how to securely outsource group exponentiation to two, possibly dishonest, servers that are physically separated (and do not communicate). Their protocol achieves security as long as one of them is honest (even if the computationally limited device does not know which one). In 2012, Chen, Li, Ma, Tang and Lou [12] presented a nice efficiency improvement to the protocol from [25], but the security of their scheme also relies on the assumption that the two servers cannot communicate.

Constructions for Outsourcing Fixed Base Exponentiation						
u	a	u^a	# delegations	Lower bound	Achieved complexity	Optimality
Public	Public	Public				
Public	Public	Secret				
Public	Secret	Public	1	0	0 (Protocol 2)	✓
Public	Secret	Secret	1	1	1 (Protocol 3)	✓
Secret	Public	Public				
Secret	Public	Secret	1	1	1 (Protocol 1)	✓
Secret	Secret	Public	1	0	0 (Protocol 2)	✓
Secret	Secret	Secret	1	1	1 (Protocol 1)	✓

Constructions for Outsourcing Variable Base Exponentiation						
u	a	u^a	# delegations	Lower bound	Achieved complexity	Optimality
Public	Public	Public				
Public	Public	Secret				
Public	Secret	Public	s	$\frac{\log p}{s+1}$	$\frac{\log p}{s+1}$ (Protocols 4&5)	✓
Public	Secret	Secret	s	$\frac{\log p}{s+1}$	$\frac{\log p}{s+1}$ (Protocols 4&5)	✓
Secret	Public	Public				
Secret	Public	Secret	1	$\frac{\log p}{2\ell+4}$	$\frac{\log p}{\ell}$ (Protocol 8)	✗
			2	≤ 3	3 (Protocol 7)	✗
Secret	Secret	Public	s	$\frac{\log p}{s+1}$	$\frac{\log p}{s}$ (Protocol 6)	✗
Secret	Secret	Secret	s	$\frac{\log p}{s+1}$	$\frac{\log p}{s}$ (Protocol 6)	✗

ℓ is the number of available pairs (k, g^k), p is the order of $\mathbb{G}$.

Table 1 Outsourcing Protocols for Single Exponentiation (Summary)

Since this separation of the two servers is actually a strong assumption hard to be met in practice, Wang, Wu, Wong, Qin, Chow, Liu and Tan [45] and independently Ding, Xu, Ye and Chooin [20] proposed some protocols to outsource group exponentiations to a *single* untrusted server. Their generic algorithms are very efficient and allow to outsource multi-exponentiations with fixed or variable exponent and bases (that can be public or secret).

1.2 Contributions of the paper

Our contributions are both theoretical and practical.

Our first result is some practical attacks on the aforementioned protocols for outsourcing (multi-)exponentiations proposed by Wang *et al.* [45] and Ding *et al.* [20]. Our attacks allow to recover secret information in polynomial time using lattice reduction. It shows that these solutions are completely insecure. In this paper, we also show that what they expected to achieve is actually theoretically impossible.

Our second contribution is the proposal of a taxonomy of exponentiation delegation protocols and the associated formal models of protocols that allow a client $\mathcal{C}$ who wants to compute a multi-exponentiation (which is a computation of the form $\prod_{i=1}^{t} u_i^{a_i}$ for group elements u_i's and exponents a_i's) to delegate an intermediate exponentiation to a more powerful server $\mathcal{S}$. The client's contribution in the computation is then only few multiplications of group elements and arithmetic operations modulo the underlying group order. We consider in this work only prime-order groups.

Our taxonomy covers all the practical situations : the group elements can be secret or public, variable or fixed, the exponents can be secret or public, and the result of the multi-exponentiation can also be either public or secret. As an example, a Boneh-Lynn-Shacham digital signature [6, 7] is a group element $\sigma = h(m)^a$, where m is the signed message, h a cryptographic hash function, and a the secret key. The signature computation can be delegated with our protocol for a public group element (the hashed value of the message), a secret exponent (the secret key), and a public output (the signature). During an ElGamal decryption of a ciphertext $(c_1, c_2) = (g^r, m \cdot y^r)$ (where m is the plaintext and $y = g^a$ is the public key), one may want to securely delegate the computation of ${c_1}^a$ (to recover m as $c_2/{c_1}^a$). Such an exponentiation can be delegated with our protocol for known group element (c_1), secret exponent (a) and secret result (${c_1}^a$, in order to keep the plaintext m secret).

We propose a delegation protocol for each of the previously mentioned scenarios. The latency of sending messages back and forth has been shown to often be the dominating factor in the running time of cryptographic protocols. Indeed, round complexity has been the subject of a great deal of research in cryptography. We thus focus on the problem of constructing one-round delegation protocols; i.e., we authorize the client to call only once the server $\mathcal{S}$, and give him access to some precomputations (consisting of pairs of the form (k, g^k)). We then consider their complexity, in terms of the number of group operations needed by the client to eventually get the desired result securely. These algorithms are simple and we prove that they are essentially optimal.

Our third and main contribution is the computation of lower bounds on the number of group operations needed on the client's side to securely compute his exponentiation when it has access to a helper server. To give these lower bounds, we analyze the security of delegation protocols in the generic group model which considers that algorithms do not exploit any properties of the encodings of group element. This model is usually used to rule out classes of attacks by an adversary trying to break a cryptographic assumption. We use it only to prove our lower bounds but we do not assume that an adversary against our protocols is limited to generic operations in the underlying group. As mentioned above, these lower

bounds tell us that our protocols cannot be significantly improved. A summary of our results is given in Table 1 (and all our results are collected in Table 2 in the core of the paper).

The rest of the paper is organized as follows: we describe the general background necessary all along the paper in Sections 2 and 3. Section 4 presents our attacks against Wang *et al.*'s and Ding *et al.*'s protocols. Section 5 contains our generic constructions of protocol for privately outsource exponentiation, Section 6 gives our lower bounds for one-round protocols, meaning that the client sends all the data to the server in one communication and the one after the lower bounds for two-round protocols. Section 7 extends our techniques to multi-round protocols. Section 8 presents generic protocols for privately outsource multi-exponentiation. The last section is our conclusion.

2 Exponentiation Delegation: Definitions

The (multi-)exponentiations are computed in a group $\mathbb{G}$ whose description is provided by an algorithm $\mathsf{GroupGen}$, which takes as input a security parameter λ. It provides a string params which contains the group description and its prime[1] order, say p. Let n be an integer, we denote by $\mathbf{a} = (a_1, \ldots, a_n)$ (*resp.* $\mathbf{u} = (u_1, \ldots, u_n)$) a vector of n exponents $a_i \in \mathbb{Z}_p$ (*resp.* group elements $u_i \in \mathbb{G}$) for $i \in \{1, \ldots, n\}$. The aim of the protocols that follow is to compute $\prod_{i=1}^{n} u_i^{a_i}$, denoted as $\mathbf{u}^{\mathbf{a}}$.

We consider a delegation of an exponentiation as a 2-party protocol between a client $\mathcal{C}$ and a server $\mathcal{S}$. We denote as $(y_{\mathcal{C}}, y_{\mathcal{S}}, tr) \leftarrow (\mathcal{C}(1^{\lambda}, \mathsf{params}, (\mathbf{a}, \mathbf{u})), \mathcal{S}(1^{\lambda}))$ the protocol at the end of which $\mathcal{C}$ knows $y_{\mathcal{C}}$ and $\mathcal{S}$ learns $y_{\mathcal{S}}$ (usually an empty string). The variable λ is a positive integer called the *security parameter*, the string tr is the transcript of the interaction. In all our protocols, the server will be very basic, since it will only perform exponentiations whose basis and exponent are sent to him by the client. In [11], Cavallo *et al.* emphasized the need for delegation of group inverses since almost all known protocols for delegated exponentiation do require inverse computations from the client. They presented an efficient and secure protocol for delegating group inverses. However, our protocols do not require such computations and our lower bounds hold even in groups in which inverse computation is efficient (and therefore does not need to be delegated, see Remark 6).

To model the security notions, and to simplify the exposition, we describe by a *computation code* $\boldsymbol{\beta}$ (which is a binary vector of length 4), the scenario of the computation. Indeed, according to the applications, some of the data on which the computations are performed may be either public or secret. In the computation of $\mathbf{u}^{\mathbf{a}}$, the vector of basis $\mathbf{u}$, the vector of exponents $\mathbf{a}$ or the result $\mathbf{u}^{\mathbf{a}}$ may be unknown (and especially to the adversary). The three first entries of the code describe the secrecy of respectively $\mathbf{u}$, $\mathbf{a}$ and $\mathbf{u}^{\mathbf{a}}$: a 0 means that the data is hidden to the adversary, and 1 means that the data is public. The last entry indicates whether the base if fixed (f) or variable (v). For instance, the code 101v means that $\mathbf{u}$ is public, the exponent $\mathbf{a}$ is secret, and the result $\mathbf{u}^{\mathbf{a}}$ is public, while the base is variable. Note that we consider the *whole* vectors $\mathbf{u}$, $\mathbf{a}$ and $\mathbf{u}^{\mathbf{a}}$ (*i.e.,* all of their coordinates) to be either public or private, whereas we could imagine that, for a

[1] In this paper, following prior works, we consider only (known) prime order groups, but some of our results can be generalized to composite order groups and unknown order groups.

vector $\mathbf{u}$ of exponents for instance, some of these could be public, and others could be kept secret. The following security notions should then be adapted according to these scenarios.

2.1 Correctness

The correctness requirement for delegation of a (multi-)exponentiation means that when the server and the client follow honestly the protocol, the client's output is actually the expected (multi-)exponentiation.

Definition 1 (Correctness) Let λ be a positive integer. We say that $(\mathcal{C}, \mathcal{S})$ satisfies *correctness* if

$$\Pr\left[y_{\mathcal{C}} = \mathbf{u}^{\mathbf{a}} = \prod_{i=1}^{n} u_i^{a_i} \;\middle|\; \begin{array}{l} \mathsf{params} = (\mathbb{G}, p) \leftarrow \mathsf{GroupGen}(1^\lambda), \\ \mathbf{u} \xleftarrow{\$} \mathbb{G}^n, \mathbf{a} \xleftarrow{\$} \mathbb{Z}_p^n, \\ (y_{\mathcal{C}}, y_{\mathcal{S}}, tr) \leftarrow (\mathcal{C}(1^\lambda, \mathsf{params}, \mathbf{a}, \mathbf{u}), \mathcal{S}(1^\lambda)) \end{array} \right] = 1.$$

2.2 Instance-Hiding

The most natural security notion that a delegation protocol must fulfill is the *instance-hiding* property. It basically means that an attacker cannot compute any secret data involved during the computation. More precisely, Fig. 1 describes the instance-hiding security experiment. The attacks presented in Section 4 break the instance-hiding property of the schemes. The attacker $\mathcal{A}$ is initially fed with information that depends on the scenario. The role of the procedure $\mathcal{I}$ is to set the initial information given to the attacker. It takes as input $\mathbf{u}$, $\mathbf{a}$ and the computation code $\boldsymbol{\beta} = (\beta_1, \beta_2, \beta_3, \beta_4)$ and outputs a subset $\mathcal{I}(\mathbf{u}, \mathbf{a}, \boldsymbol{\beta}) \subseteq \{\mathbf{u}, \mathbf{a}, \mathbf{u}^{\mathbf{a}}\}$ such that

- $\mathbf{u} \in \mathcal{I}(\mathbf{u}, \mathbf{a}, \boldsymbol{\beta})$ if and only if $\beta_1 = 1$;
- $\mathbf{a} \in \mathcal{I}(\mathbf{u}, \mathbf{a}, \boldsymbol{\beta})$ if and only if $\beta_2 = 1$;
- $\mathbf{u}^{\mathbf{a}} \in \mathcal{I}(\mathbf{u}, \mathbf{a}, \boldsymbol{\beta})$ if and only if $\beta_3 = 1$.

The attacker then engages in a series of delegation protocols, where he can adaptively choose the secrets involved during the protocols, including the target ones (by setting a Boolean flag $\mathsf{challenge}$ to `true`), and he eventually outputs an answer $A^\star = (\mathbf{u}^\star, \mathbf{a}^\star, v^\star) \in \mathbb{G}^n \times \mathbb{Z}_p^n \times \mathbb{G}$ (in the final delegation protocol when another flag α is set to `attack`). The attacker is said to win this experiment, if the predicate $P_1(A^\star, \mathbf{u}, \mathbf{a})$ holds where $P_1(A^\star, \mathbf{u}, \mathbf{a})$ is equal to 1 if and only if the three following equalities hold:

- $\mathbf{u}^\star = \mathbf{u}$;
- $\mathbf{a}^\star = \mathbf{a}$;
- $v^\star = \mathbf{u}^{\mathbf{a}}$.

Note that if the computation code $\boldsymbol{\beta}$ contains some values equal to 1, it is actually trivial for the adversary to output an answer $A^\star$ that satisfies some of these equalities. In particular, for the case where $\mathbf{u}$, $\mathbf{a}$, $\mathbf{u}^{\mathbf{a}}$ are public (i.e., the computation code $\boldsymbol{\beta} = (1, 1, 1, \beta_4)$ for any $\beta_4 \in \{f, v\}$), the security notion cannot be achieved. Similarly, the security notion for the cases with the computation code $\boldsymbol{\beta} = (1, 1, 0, \beta_4)$ and $\boldsymbol{\beta} = (0, 1, 1, \beta_4)$ (for any $\beta_4 \in \{f, v\}$) cannot be achieved (but the latter case may have some interest for composite order groups).

Experiment $\mathbf{Exp}_{ih}(\mathcal{A}, \boldsymbol{\beta}, \lambda)$
$\mathsf{params} = (\mathbb{G}, p) \leftarrow \mathsf{GroupGen}(1^\lambda)$
$(\mathbf{a}, \mathbf{u}) \xleftarrow{R} \mathbb{Z}_p^n \times \mathbb{G}^n$
$init \leftarrow \mathcal{I}(\mathbf{u}, \mathbf{a}, \boldsymbol{\beta})$
$i \leftarrow 1$, $tr_0 \leftarrow \emptyset$
$(\alpha, \mathsf{challenge}, aux, (\mathbf{a}_1, \mathbf{u}_1)) \leftarrow \mathcal{A}(1^\lambda, \mathsf{params}, init)$
while $\alpha \neq$ `attack` do
  if $\mathsf{challenge} =$ `true` do
    $(y_i, (\alpha, \mathsf{challenge}, (\mathbf{a}_{i+1}, \mathbf{u}_{i+1}), aux), tr_i) \leftarrow (\mathcal{C}(1^\lambda, \mathsf{params}, (\mathbf{a}, \mathbf{u})), \mathcal{A}(aux))$
  otherwise do
    $(y_i, (\alpha, \mathsf{challenge}, (\mathbf{a}_{i+1}, \mathbf{u}_{i+1}), aux), tr_i) \leftarrow (\mathcal{C}(1^\lambda, \mathsf{params}, (\mathbf{a}_i, \mathbf{u}_i)), \mathcal{A}(aux))$
  $i \leftarrow i + 1$
$(y, A^\star, tr) \leftarrow (\mathcal{C}(1^\lambda, \mathsf{params}, (\mathbf{a}, \mathbf{u})), \mathcal{A}(aux))$
Return 1 if $P_1(A^\star, \mathbf{u}, \mathbf{a}) = 1$ and 0 otherwise

Fig. 1 Instance-hiding

Definition 2 (Instance-hiding) Let $n > 0$ be some integer, $\mathsf{GroupGen}$ be a group generator, and $(\mathcal{C}, \mathcal{S})$ be a client-server protocol for the server-aided computation of the n-ary multi-exponentiation for $\mathsf{GroupGen}$. Let $\boldsymbol{\beta} \in \{0,1\}^4$ be a computation code and let $\tau : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$. be two functions. We say that $(\mathcal{C}, \mathcal{S})$ satisfies (τ, ε)-*instance-hiding* if, for any algorithm $\mathcal{A}$, it holds that for all integer $\lambda \in \mathbb{N}$

$$\Pr[\nu = 1 | \nu \leftarrow \mathbf{Exp}_{ih}(\mathcal{A}, \boldsymbol{\beta}, \lambda)] \leq \varepsilon(\lambda)$$

where $\mathbf{Exp}_{ih}(\mathcal{A}, \boldsymbol{\beta}, \lambda)$ is the instance-hiding computational random experiment described in Figure 1 in which $\mathcal{A}$ runs in time at most $\tau(\lambda)$.

We can consider two variants of the instance-hiding security notion: the weakest one considers an *honest-but-curious* adversary $\mathcal{A}$ (which follows the delegation protocol executions honestly but hopes to learn information from them) and the stronger notion considers a *malicious* adversary $\mathcal{A}$ (who can deviate arbitrarily from the specified protocol execution).

Remark 1 In the instance-hiding computational random experiment $\mathbf{Exp}_{ih}(\mathcal{A}, \boldsymbol{\beta}, \lambda)$, the adversary can run arbitrarily many delegation protocols with the challenger (for a total running time upper-bounded by $\tau(\lambda)$) , either on inputs of its choice $(\mathbf{a}_i, \mathbf{u}_i)$ (when $\mathsf{challenge} =$ `false`) or on the challenge input $(\mathbf{a}, \mathbf{u})$ (when $\mathsf{challenge} =$ `true`). Eventually, the adversary sets $\alpha =$ `attack`, and at the end of a final execution of the delegation protocol on the challenge input $(\mathbf{a}, \mathbf{u})$, it outputs $A^\star = (\mathbf{u}^\star, \mathbf{a}^\star, v^\star) \in \mathbb{G}^n \times \mathbb{Z}_p^n \times \mathbb{G}$ and succeeds if the equality $P_1(A^\star, \mathbf{u}, \mathbf{a}) = 1$ holds.

2.3 Indistinguishability

We describe now a notion of security that relaxes the usual simulation-based security from [25] and [45]. The simulation-based security notion captures in perhaps the most direct way the intuition of a good notion of privacy. Roughly, it says that "whatever can be efficiently computed about the secret inputs given the protocol's view can be computed without this view". However, it is a relatively complex and subtle notion to formalize (see [25] or [45] for details).

In this paper we instead consider a simpler indistinguishability-based security notion that captures that an untrusted helper cannot tell which inputs the other parties might have used. The formalization was provided in [11]. It is simple and easy to use: it says that if we take two secret inputs (even adversarially chosen), an adversary running the outsource protocol with one input picked uniformly at random cannot tell which it was with a probability significantly better than that of guessing. This notion is similar to *Input-Indistinguishable Computation* introduced by Micali, Pass and Rosen in [33].

Note that this security notion is implied by the simulation-based one from [25, 45]. In particular, since we will prove that the protocol from [45] does not achieve our security notion, we obtain that it does not achieve the stronger simulation-based security notion from [25,45] (contrary to what is claimed in [45]). In Section 6, we prove that it is impossible to design some secure outsourcing exponentiation protocols (for our security definition) for a single untrusted computational resource if one wants to limit the computational cost of the limited device to a constant number of (generic) group operations. This result readily implies that this task is also impossible for the stronger simulation-based security notion from [25] and [45].

Once again, the advantage of the adversary $\mathcal{A}$ in the indistinguishability experiment, depicted in Fig. 2, will be settled according to the context of the delegation. We use the predicate $P_2((\mathbf{a}_0^\star, \mathbf{u}_0^\star), (\mathbf{a}_1^\star, \mathbf{u}_1^\star), \boldsymbol{\beta})$ to tell that when the base, the exponent or the result of the exponentiation is known (according to the computation code), the base, the exponent chosen by the adversary or the corresponding result must be the same for both pairs (otherwise, the privacy would be trivially broken). The predicate $P_2((\mathbf{a}_0^\star, \mathbf{u}_0^\star), (\mathbf{a}_1^\star, \mathbf{u}_1^\star), \boldsymbol{\beta})$ is defined as the conjunction of the three following disjunctions:

- $\mathbf{u}_0^\star = \mathbf{u}_1^\star$ or $\beta_1 = 0$;
- $\mathbf{a}_0^\star = \mathbf{a}_1^\star$ or $\beta_2 = 0$;
- $(\mathbf{u}_0^\star)^{\mathbf{a}_0^\star} = (\mathbf{u}_1^\star)^{\mathbf{a}_1^\star}$ or $\beta_3 = 0$;

As above, the security definition cannot be achieved for some computation codes.

Definition 3 (Indistinguishability) Let $n > 0$ be some integer, $\mathsf{GroupGen}$ be a group generator, and $(\mathcal{C}, \mathcal{S})$ be a client-server protocol for the server-aided computation of the n-ary multi-exponentiation for $\mathsf{GroupGen}$. Let $\boldsymbol{\beta} \in \{0,1\}^4$ be the computation code and let $\tau : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$. be two functions. We say that $(\mathcal{C}, \mathcal{S})$ satisfies (τ, ε)-*indistinguishability* if, for any algorithm $\mathcal{A}$, it holds that for all integer $\lambda \in \mathbb{N}$

$$\left| \Pr[\nu = 1 | \nu \leftarrow \mathbf{Exp}_{ind}(\mathcal{A}, \boldsymbol{\beta}, \lambda)] - \frac{1}{2} \right| \leq \varepsilon(\lambda)$$

where $\mathbf{Exp}_{ind}(\mathcal{A}, \boldsymbol{\beta}, \lambda)$ is the indistinguishability computational random experiment described in Figure 2 in which $\mathcal{A}$ runs in time at most $\tau(\lambda)$.

As above, we can consider two variants of the indistinguishability security notion with an *honest-but-curious* adversary $\mathcal{A}$ or a *malicious* adversary $\mathcal{A}$.

Remark 1 *As mentioned in [10, 14, 23], a delegation protocol that does not ensure verifiability may cause severe security problems (in particular if the delegated computation occurs in the verification algorithm of some authentication protocol). However, verifiability is not necessarily mandatory in scenarios where the delegated computation is used*

Experiment $\mathbf{Exp}_{ind}(\mathcal{A}, \boldsymbol{\beta}, \lambda)$
$\mathsf{params} = (\mathbb{G}, p) \leftarrow \mathsf{GroupGen}(1^\lambda)$
$(\alpha, (\mathbf{a}_1, \mathbf{u}_1), aux) \leftarrow \mathcal{A}(1^\lambda, \mathsf{params})$
$i \leftarrow 1$, $tr_0 \leftarrow \emptyset$
while $\alpha \neq \texttt{attack}$ do
$(y_i, (\alpha, (\mathbf{a}_{i+1}, \mathbf{u}_{i+1}), aux), tr_i) \leftarrow (\mathcal{C}(1^\lambda, \mathsf{params}, (\mathbf{a}_i, \mathbf{u}_i)), \mathcal{A}(aux))$
$i \leftarrow i + 1$
$((\mathbf{a}_0^\star, \mathbf{u}_0^\star), (\mathbf{a}_1^\star, \mathbf{u}_1^\star), aux) \leftarrow \mathcal{A}(aux)$
$b \xleftarrow{\$} \{0, 1\}$
$(y, b^\star, tr) \leftarrow (\mathcal{C}(1^\lambda, \mathsf{params}, (\mathbf{a}_b^\star, \mathbf{u}_b^\star)), \mathcal{A}(aux))$
Return 1 if $(b^\star = b) \wedge P_2((\mathbf{a}_0^\star, \mathbf{u}_0^\star), (\mathbf{a}_1^\star, \mathbf{u}_1^\star), \boldsymbol{\beta})$
0 otherwise

Fig. 2 Indistinguishability

for instance in an encryption scheme as a session key. In this case, one can indeed use additional cryptographic techniques to ensure that the values returned by the powerful device are correct (e.g. by adding a MAC or other redundancy to the ciphertext).

3 Underlying Tools

3.1 Generic Group Model

Let GroupGen be a group generator. As mentioned above, it takes as input a security parameter λ and provides a set params which contains a description of a (multiplicative) group $(\mathbb{G}, \cdot)$, the group order, say $p = |\mathbb{G}|$, and one generator g. As usual, the generic group model in $\mathbb{G}$ is implemented by choosing a random injective encoding $\sigma : \mathbb{G} \longrightarrow \{0,1\}^m$ (with $2^m > p$). Instead of working directly with group elements, a generic algorithm $\mathcal{A}$ takes as input (in addition to the group order p) their image under σ. This way, all $\mathcal{A}$ can test is group elements equality (by encoding equality). $\mathcal{A}$ is also given access to an oracle $\mathcal{G}$ computing group multiplication: taking two encodings $\sigma(g_1)$ and $\sigma(g_2)$ of two group elements $g_1, g_2 \in \mathbb{G}$ as inputs and returning the encoding $\sigma(g_1 \cdot g_2)$ of the product $g_1 \cdot g_2 \in G$. We can assume that $\mathcal{A}$ submits to the oracle only encodings of elements it had previously received.

This is because we can choose m large enough so that the probability of choosing a string that is also in the image of σ is negligible (see [42] for details). In particular, in this paper[2], a generic algorithm $\mathcal{A}$ cannot generate encodings of new group elements.

Usually, the generic group model is used to rule out classes of attacks by an adversary trying to break a cryptographic assumption. In contrast, in this paper, we use the generic group model to prove a lower bound on the complexity of the delegation protocol. In order to prove our complexity lower bounds, we make intensive use of the following simple lemma:

Lemma 1 *Let* GroupGen *be a group generator, let* $\mathbb{G}$ *be a group of prime order* p *output by* GroupGen *and let* $\mathcal{A}$ *be a generic algorithm in* $\mathbb{G}$*. If* $\mathcal{A}$ *is given as inputs encodings*

[2] The lower bounds on the complexity of generic delegation protocols given in Section 6 and Section 7 do hold without this assumption but with unnecessarily complicated proofs.

$\sigma(g_1), \ldots, \sigma(g_n)$ of group elements $g_1, \ldots, g_n \in \mathbb{G}$ (for $n \in \mathbb{N}$) and outputs the encoding $\sigma(h)$ of a group element $h \in \mathbb{G}$ in time τ, then there exist positive integers $\alpha_1, \ldots, \alpha_n$ such that $h = g_1^{\alpha_1} \ldots g_n^{\alpha_n}$ and $\max(\alpha_1, \ldots, \alpha_n) \leq 2^\tau$.

Note that in the statement of Lemma 1, τ is an upper-bound on the total running time of $\mathcal{A}$ but the result holds also with τ an upper-bound on the query complexity of $\mathcal{A}$ to the group oracle $\mathcal{G}$.

Proof We can define a map $\pi : \{0,1\}^m \to \mathbb{Z}^n$ which associates to each encoding obtained by $\mathcal{A}$ during its execution an n-dimensional vector in $\mathbb{Z}^n$. For each input encoding $\sigma(g_i)$, $\pi(\sigma(g_i))$ is defined as the i-th vector from the $\mathbb{Z}^n$ canonical basis (for $i \in \{1, \ldots, n\}$) and for each encoding $\sigma(h_1)$ and $\sigma(h_2)$ queried to $\mathcal{G}$, $\pi(\sigma(h_1 \cdot h_2)) = \pi(\sigma(h_1)) + \pi(\sigma(h_2))$. By construction, during the whole execution of $\mathcal{A}$, we have $\pi(\sigma(h)) = (\alpha_1, \ldots, \alpha_n)$ if and only if $h = g_1^{\alpha_1} \ldots g_n^{\alpha_n}$ for all encodings $\sigma(h)$. Moreover, during the computation, the ℓ_∞-norm of $\pi(\sigma(h_1 \cdot h_2))$ is upper-bounded by $\ell_\infty(\pi(\sigma(h_1))) + \ell_\infty(\pi(\sigma(h_2)))$. Since the ℓ_∞-norm of the input encodings $\pi(\sigma(g_i))$ is equal to 1 (for $i \in \{1, \ldots, n\}$) and the ℓ_∞-norm of encodings at most doubles for each query to $\mathcal{G}$, we obtained the claimed result.

For generic algorithm that takes as inputs encodings of group elements in different phases using some oracle, one can easily generalize the previous lemma.

Lemma 2 *Let* GroupGen *be a group generator, let $\mathbb{G}$ be a group of prime order p output by* GroupGen *and let $\mathcal{A}$ be a generic algorithm in $\mathbb{G}$ with oracle access. We suppose that $\mathcal{A}$ makes $k \in \mathbb{N}$ queries to some oracle that returns encodings of group elements:*

- *$\mathcal{A}$ gets inputs encodings $\sigma(g_{0,1}), \ldots, \sigma(g_{0,n_0})$ of group elements $g_{0,1}, \ldots, g_{0,n_0} \in \mathbb{G}$ (for $n_0 \in \mathbb{N}$)*
- *after computation time τ_1, $\mathcal{A}$ gets encodings $\sigma(g_{1,1}), \ldots, \sigma(g_{1,n_1})$ of group elements $g_{1,1}, \ldots, g_{1,n_1} \in \mathbb{G}$ (for $n_1 \in \mathbb{N}$)*
- *...*
- *after computation time τ_k, $\mathcal{A}$ gets encodings $\sigma(g_{k,1}), \ldots, \sigma(g_{k,n_k})$ of group elements $g_{k,1}, \ldots, g_{k,n_k} \in \mathbb{G}$ (for $n_k \in \mathbb{N}$)*

with $\tau_1 \leq \cdots \leq \tau_k$. If $\mathcal{A}$ runs in total time $\tau \geq \tau_k$ and outputs the encoding $\sigma(h)$ of a group element $h \in \mathbb{G}$ in time τ, then there exist positive integers

$$\alpha_{0,1}, \ldots, \alpha_{0,n_0}, \ldots, \alpha_{k,1}, \ldots, \alpha_{k,n_k}$$

such that

$$h = (g_{0,1}^{\alpha_{0,1}} \ldots g_{0,n_0}^{\alpha_{0,n_0}}) \cdot (g_{1,1}^{\alpha_{1,1}} \ldots g_{1,n_1}^{\alpha_{1,n_1}}) \cdots (g_{k,1}^{\alpha_{k,1}} \ldots g_{k,n_k}^{\alpha_{k,n_k}})$$

and $\max(\alpha_{0,1}, \ldots, \alpha_{0,n_0}) \leq 2^\tau$ and $\max(\alpha_{i,1}, \ldots, \alpha_{i,n_i}) \leq 2^{\tau - \tau_i}$ for $i \in \{1, \ldots, k\}$.

Proof The proof is by induction on k. The basic step for $k = 0$ is simply Lemma 1. The inductive step is readily obtained following the proof of Lemma 1.

All the exponentiation delegation protocols we present in Section 5 are generic (interactive) algorithms. However, we want to stress that their security analysis is provided in the standard security model (without any idealized assumption and in particular we do not assume that an adversary is limited to generic operations in the underlying group).

In some specific groups, it is possible to improve the efficiency of exponentiation algorithms by using non-generic operations (see [21, § 11.3], for instance):

- One may take advantage of an additional structure in subgroups of the multiplicative group of (non-prime) finite fields[3] $\mathbb{F}_{q^n}^*$ with $n \geq 2$. Indeed, in this setting one can use a *normal basis* $\{\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}\}$ of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ to represent group elements in order to make the computation of the q-th power of an element as a simple (and almost free) cyclic shift of its representation.
- One may also take advantage of the fact that in certain algebraic groups, group inversion is sometimes more efficient than a group multiplication (in particular in subgroups of elliptic curves over finite fields). In this case, one can use signed expansions of exponents when computing (multi-)exponentiation and in particular the *w-ary non-adjacent form* method which guarantees that on average there will be fewer group multiplications in Algorithm 1 for instance (see below).
- One may use (more generally) groups equipped with efficient endomorphisms (e.g., Frobenius endomorphism, complex multiplication endomorphism). This method was originally proposed by Gallant, Lambert and Vanstone [22] to perform *group exponentiation with endomorphism decomposition*. In a cyclic group $\mathbb{G}$, any endomorphism is the group exponentiation by some integer (*eigenvalue* of the endomorphism) and for general group exponentiation, one can decompose the exponent as a weighted sums of these eigenvalues (with "small" weights) and then use a multi-exponentiation algorithm such as Algorithm 1 (see below).

It is sometimes possible to improve (but only by a constant factor) the efficiency of the exponentiation delegation protocols we present in Section 5 by using similar techniques. In these situations, the complexity lower bounds from Section 6 and Section 7 do no hold anymore but one can adapt our arguments (see Remark 6 for instance).

3.2 Multi-exponentiation by Simultaneous 2^w-ary method

Algorithm 1 computes the multi-exponentiation $\prod_{i=1}^{t} g_i^{x_i} \in \mathbb{G}$, for $g_1, \ldots, g_t \in \mathbb{G}$ and $x_1, \ldots, x_t \in \mathbb{N}$ by using the simultaneous 2^w-ary method introduced by Straus in 1964 [44]. The method looks at w bits of each of the exponents for each evaluation stage group multiplication (where w is a small positive integer), i.e. tw bits in total (see [3, 34] for details of different multi-exponentiation techniques).

Complexity: The precomputed table contains $2^{tw} - 1 - t$ non-trivial entries among which $2^{t(w-1)} - 1$ can be computed by squaring other table entries (all the E_i are even). The remaining $2^{tw} - 2^{t(w-1)} - t$ entries require one general multiplication each. The total cost is for the precomputation phase $2^{tw} - 2^{t(w-1)} - t$ multiplications and $2^{t(w-1)} - 1$ squarings and $\ell(2^t w - 1)/2^{tw} w \leq \ell/w$ multiplications on average and ℓ squarings. For $t = 2$, the cost is minimal for w around $1/2 \log \ell - \log\log \ell$ with $\ell(1 + 3/\log \ell) = \ell(1 + o(1))$ multiplications overall.

[3] The most studied case was $q = 2$ but it is not interesting anymore in cryptography due to the recent impressive progress on finite field discrete logarithms [5]. However, this technique may still found applications in pairing-based cryptography.

Algorithm 1 Multi-Exponentiation by Simultaneous 2^w-ary method

Input: $g_1, \ldots, g_t \in \mathbb{G}$, $x_1, \ldots, x_t \in \mathbb{N}$ with $\ell = \max_{i \in \{1,\ldots,t\}} \lceil \log x_i \rceil$ and $x_j = \sum_{i=0}^{\lfloor \ell/w \rfloor - 1} e_{i,j} 2^{wi} \in \mathbb{N}$ and $e_{i,j} \in \{0, 2^w - 1\}$ for $i \in \{0, \ldots, \ell/w - 1\}$ and $j \in \{1, \ldots, t\}$
Output: $g_1^{x_1} \cdots g_t^{x_t} \in \mathbb{G}$
  for all non-zero t-tuples $E = (E_1, \ldots, E_t) \in \{0, \ldots, 2^w - 1\}^t$ **do**
    $g_E \leftarrow \prod_{1 \le i \le t} g_i^{E_i}$ ▷ Precomputation stage
  end for
  $h \leftarrow 1_{\mathbb{G}}$
  for i from $\lfloor \ell/w \rfloor - 1$ to 0 **do**
    $h \leftarrow h^{2^w}$
    $E \leftarrow (e_{i,1}, e_{i,2}, \ldots, e_{i,t})$
    $h \leftarrow h \cdot g_E$ ▷ Multiply h by table entry $g_E = \prod_{1 \le k \le t} g_i^{e_{i,k}}$
  end for
  return h

3.3 Decomposition of Exponents

Let p be a prime number (in our protocols in the following, p will be the order of the underlying group $\mathbb{G}$). Let $s \geq 1$ be an integer and $\rho_1, \ldots, \rho_s \in \mathbb{Z}_p$. Let $a \in \mathbb{Z}_p$, an *s-dimensional decomposition of a with respect to $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_s)$* is an s-dimensional vector $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_s) \in \mathbb{Z}_p^s$ such that

$$\langle \boldsymbol{\alpha}, \boldsymbol{\rho} \rangle := \alpha_1 \rho_1 + \cdots + \alpha_s \rho_s = a \bmod p.$$

It is well-known that if the scalars ρ_i for $i \in \{1, \ldots, s\}$ have pairwise differences of absolute value at least $p^{1/s}$, then there exists a polynomial-time algorithm which on inputs a and $\boldsymbol{\rho}$ outputs a s-dimensional decomposition $\boldsymbol{\alpha} \in \mathbb{Z}_p^s$ of a with respect to $\boldsymbol{\rho}$ such that $0 \leq \alpha_i \leq C \cdot p^{1/s}$ for $i \in \{1, \ldots, s\}$ (for some small constant $C > 0$). To find this "small decomposition" of a, the algorithm applies a lattice reduction algorithm (such as the LLL-algorithm) to produce a short basis of the $\mathbb{Z}$-lattice of dimension $s+1$ spanned by the vectors $(p, 0, 0, \ldots, 0)$, $(\rho_1, 1, 0, \ldots, 0)$, $(\rho_2, 0, 1, \ldots, 0)$, $\ldots$, $(\rho_s, 0, 0, \ldots, 1)$ and applies Babai rounding algorithm [4] to find a nearby vector in this lattice from $(a, 0, \ldots, 0)$ (see [43] for details). In the following, we will refer to this algorithm as the GLV Decomposition Algorithm (GLV-Dec for short) since the method was first introduced by Gallant, Lambert and Vanstone [22] to perform group exponentiations with endomorphism decomposition.

Many important problems in cryptanalysis amount to solving polynomial equations with partial information about the solutions. In 1996, Coppersmith introduced two celebrated lattice-based techniques [15, 16] for finding small roots of polynomial equations. In the following, we will consider settings in which there exists an s-dimensional decomposition of a scalar a that is significantly shorter that the one produced by the GLV Decomposition Algorithm. Given some bounds $X_1, \ldots, X_s$ in $\mathbb{N}$ such that $X_1 \cdots X_s < p$, for a random scalar $a \in \mathbb{Z}_p$ and a random vector $\boldsymbol{\rho} \in \mathbb{Z}_p^s$, we expect a unique vector $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_s) \in \mathbb{Z}_p^s$ such that $\langle \boldsymbol{\alpha}, \boldsymbol{\rho} \rangle = a$ with $\alpha_i < X_i$ for all $i \in \{1, \ldots, s\}$. In Section 4, we provide an algorithm that solves such a problem for $s = 2$. In his PhD thesis [24], Herrmann mentions a "folklore method" to solve this problem:

Method 1 *(adapted from [24, Theorem 6]) Let s be an integer. Let $p \in \mathbb{N}$ and $f(x_1, \ldots, x_s) = \rho_1 x_1 + \cdots + \rho_s x_s$ be a linear polynomial in s variables with* $\gcd(\rho_i, p) =$

1 *for at least one* $i \in \{1, \ldots, s\}$. *Further let* $X_i \in \mathbb{N}$ *be some positive integers. If there exist a vector of integers* $(\alpha_1, \ldots, \alpha_s)$ *such that* $f(\alpha_1, \ldots, \alpha_s) = 0 \mod p$ *with* $|\alpha_i| \leq X_i$ *for* $i \in \{1, \ldots, s\}$ *then, heuristically, we can find the solution* $(\alpha_1, \ldots, \alpha_s)$ *if* $\prod_{i=1}^{s} X_i \leq p$ *in time polynomial in* $\log p$ *(for a constant* s*).*

This Coppersmith-like result holds for the homogeneous case, so our exponent decomposition problem can be solved by adding an extra variable α_{s+1}. The heuristic assumption in the Method 1 comes from the fact that the lattice constructed by the algorithm may contain several solutions (bounded by the X_i's) which satisfy $f(x_1, \ldots, x_n) = 0 \mod p$ (see [24] for details). In the context of our proof of optimality, this is not an issue, since finding any solution will allow to distinguish between the challenge cases.

3.4 Computations of pairs (g^k, k).

To outsource the computation of an exponentiation in a group $\mathbb{G}$ of prime order p, (pseudo-)random pairs of the form $(g^k, k) \in \mathbb{G} \times \mathbb{Z}_p^*$ are sometimes used to hide sensitive information to the untrusted server. This looks like a "chicken-and-egg problem" but there exist several techniques to make it possible for a computationally limited device to have such pairs at its disposal, at a low cost. A trivial method is to load its memory with many genuine (generated by a trusted party) random and independent couples. In other settings, a mobile device with limited computing capabilities can precompute "offline" such pairs at low speed and power. If the device can do a little more computation, there exist other preprocessing techniques, that may depend whether the base or the exponent varies.

We only mention here the main technique to produce these pairs (among many others [9,18,31]). The key ingredient is Boyko, Peinado and Venkatesan generator from [8]: the idea is to store a small number of precomputed pairs (g^{α_i}, α_i), and when a fresh pair is needed, the device outputs a product $g^k = \prod_{i \in S} g^{\alpha_i}$ with $k = \sum_{i \in S} \alpha_i$ for a random set S. It has then been improved by Nguyen, Shparlinski and Stern generator [35], that allows to re-use some α_i in the product. This generator is secure against adaptive adversaries and performs $o(\log(p))$ group operations. For some parameters, the generator from [35] is proved to have an output distribution statistically close to the uniform distribution. Obviously, these generators are of practical interest only if the base g is fixed and used multiple times.

In the sequel we will assume that the client may have access to some *random power generator* $\mathcal{B}(\cdot)$ that at invocation (with no input) outputs a single random pair $(g^k, k) \in \mathbb{G} \times \mathbb{Z}_p^*$ where k is uniformly distributed in $\mathbb{Z}_p^*$ (or statistically close to the uniform distribution). If the generator $\mathcal{B}(\cdot)$ is invoked several times, we assume that the output pairs are independent. In order to evaluate the efficiency of delegation protocols, we consider explicitly the query complexity to the generator $\mathcal{B}(\cdot)$ (depending on the context, this can be interpreted as storage of precomputed values, offline computation or use of the generator from [35] and thus additional multiplications in $\mathbb{G}$).

4 Attacks on two delegation protocols

4.1 Attack on Wang *et al.*'s protocol [45]

Wang *et al.* proposed a generic algorithm to outsource the computation of several multi-exponentiations with variable exponents and variable bases. Their algorithm, called GExp, takes as input a list of tuples $((\{a_{i,j}\}_{1\leq j\leq s};\{u_{i,j}\}_{1\leq j\leq s}))_{1\leq i\leq r}$ and computes the list of multi-exponentiations $(\prod_{j=1}^{s} u_{i,j}^{a_{i,j}})_{1\leq i\leq r}$. It is claimed that this algorithm is secure in a strong model where the computation is outsourced to a single untrusted server [45, Theorem 1]. We will show that GExp can be broken in polynomial time using lattice reduction if two (simple) exponentiations are outsourced with the *same* exponent, which is the case in the scenario of proof of data possession presented in [45, §4]. This means that GExp does not achieve the claimed security.

Description of Wang *et al.*'s protocol. The setting of GExp is the following: $\mathbb{G}$ is a cyclic group of prime order p, and g is a generator. For $1 \leq i \leq r$ and $1 \leq j \leq s$, $a_{i,j}$ are uniform and independent elements of $\mathbb{Z}_p^*$, and $u_{i,j}$ are random elements from $\mathbb{G}$. They assume the $a_{i,j}$'s, the $u_{i,j}$'s and the result are secret (and the $u_{i,j}$ are variable, i.e. $\boldsymbol{\beta} = 000v$ with our notations).

The protocol is divided into three steps:

- **Step 1.** The client $\mathcal{C}$ generates four random pairs $(\alpha_k, \mu_k)_{1\leq k\leq 4}$ where $\mu_k = g^{\alpha_k}$ (using a random power generator). A Υ-bit element χ is randomly picked (for some parameter Υ). Then, for all $1 \leq i \leq r$ and $1 \leq j \leq s$, the elements $b_{i,j}$ are randomly picked in $\mathbb{Z}_p^*$. It sets[4]

$$c_{i,j} = a_{i,j} - b_{i,j}\chi \mod p \tag{1}$$
$$w_{i,j} = u_{i,j}/\mu_1 \tag{2}$$
$$h_{i,j} = u_{i,j}/\mu_3 \tag{3}$$
$$\theta_i = \alpha_1 \sum_{j=1}^{s} b_{i,j} - \alpha_2 + \alpha_3 \sum_{j=1}^{s} c_{i,j} - \alpha_4 \mod p. \tag{4}$$

- **Step 2.** The second step consists in invoking the (untrusted) server $\mathcal{S}$ for some exponentiations. To do so, $\mathcal{C}$ generates (using a random power generator) $r+2$ random pairs $(g^{t_i}, t_i)_{1\leq i\leq r+2}$ and queries (in random order) $\mathcal{S}$ on
  - $(g^{t_i}, \theta_i/t_i)$ to obtain $B_i = g^{\theta_i}$ for all $1 \leq i \leq r$,
  - $(g^{t_{r+1}}, \theta/t_{r+1})$ to obtain $A = g^{\theta}$ with $\theta = t_{r+2} - \sum_{i=1}^{r} \theta_i \mod p$,
  - $\begin{cases} (w_{i,j}, b_{i,j}) \text{ to get } C_{i,j} = (u_{i,j}/\mu_1)^{b_{i,j}} \\ (h_{i,j}, c_{i,j}) \text{ to get } D_{i,j} = (u_{i,j}/\mu_3)^{c_{i,j}} \end{cases}$
    for $1 \leq i \leq r$ and $1 \leq j \leq s$.
- **Step 3.** It consists in combining the different values obtained from $\mathcal{S}$ to recover the desired multi-exponentiations. In particular, an exponentiation to the power χ is involved. The protocol to be efficient, needs χ not too large.

Simple attack. Suppose that a delegation of a single exponentiation u^a, for u and a secret, is performed using Wang *et al.*'s protocol. If a is a secret key, an

[4] Note that the protocol from [45] can also be described without inversion in the group $\mathbb{G}$ but to help the reader familiar with this paper, we use the same description.

element of the form h^a is likely to be known by the adversary, together with h (one can think of a public key in a scenario of delegation of BLS signatures [7], for instance)). In this case, as the attacker sees an element of the form $c = a - b\chi$ (see Equation (1)) and knows b (cf. Step 2), he can compute h^c which is equal to $h^a \cdot (h^\chi)^{-b}$, so that recovering χ can be done by computing the discrete logarithm of $(h^a/h^c)^{b^{-1}}$ in base h. Using a baby-step giant-step algorithm, this can be done in $2^{\Upsilon/2}$ operations, which contradicts [45, Theorem 1].

Main attack. The crucial weakness of this protocol is the use of this *small* element χ which hides the exponents. The authors suggest to take it of bit-size Υ, for $\Upsilon = 64$. We will show that it cannot be that small since it can be recovered if the client outsources two exponentiations with the *same* exponent to the server $\mathcal{S}$. The scenario of our attack is the following: two exponentiations of the form $\mathsf{GExp}((a_{1,1}, \ldots, a_{1,s}); (u_{1,1}, \ldots, u_{1,s}))$ and $\mathsf{GExp}((a_{1,1}, \ldots, a_{1,s}); (u'_{1,1}, \ldots, u'_{1,s}))$ are queried to $\mathcal{S}$. The exponentiations are computed with the *same* exponents. This is typically the case in the first application proposed in [45, Section 4.1] to securely offload Shacham and Waters's proofs of retrievability [40].

For the sake of clarity, it is sufficient to focus on the elements that mask the first exponent $a_{1,1}$. An attacker will obtain (see Step 2) $b_{1,1}$, $b'_{1,1}$, $c_{1,1}$ and $c'_{1,1}$ such that $c_{1,1} = a_{1,1} - b_{1,1}\chi \mod p$ and $c'_{1,1} = a_{1,1} - b'_{1,1}\chi' \mod p$. Since the exponentiation delegated to the server are queried in a random order, the attacker has to find which queries correspond to $b_{1,1}$, $b'_{1,1}$, $c_{1,1}$ and $c'_{1,1}$ in Step 2. It can simply tries all possible 4-tuples and the complexity is multiplied by the polynomial factor $\binom{r(s+1)+1}{4}$.

Subtracting these two equations gives a modular bi-variate linear equation:

$$b_{1,1}X - b'_{1,1}Y + c_{1,1} - c'_{1,1} = 0 \mod p \tag{5}$$

which has χ and χ' as roots, satisfying $\chi \leq X$ and $\chi' \leq Y$, for some X and Y which will be larger that 2^Υ, say 2^{64}. In the following, we show that it is (heuristically) possible to recover in polynomial time any χ and χ' that are lower than $\sqrt{p}$.

Solving this bi-variate polynomial equation with small modular roots can be done using the well-known Coppersmith technique [16]. Finding small roots of modular bi-variate polynomials was studied in [28], but his method is very general, whereas we consider here only simple linear polynomials. The following lemma, inspired by Howgrave-Graham's lemma [26] suggests how to construct a particular lattice that will help to recover small modular roots of a linear polynomial in $\mathbb{Z}[x, y]$. We denote as $\|\cdot\|$ the Euclidean norm of polynomials.

Lemma 3 *Let p be a prime number and let $g(x, y) \in \mathbb{Z}[x, y]$ be a linear polynomial. If there exists four integers $X, Y \in \mathbb{N}$ and $x_0, y_0 \in \mathbb{Z}$ such that*

- $g(x_0, y_0) = 0 \mod p$,
- $|x_0| < X$ *and* $|y_0| < Y$,
- $\|g(xX, yY)\| < p/\sqrt{3}$.

Then $g(x_0, y_0) = 0$ holds over the integers.

Let us write a bi-variate linear polynomial as $P(x, y) = x + by + c$, with $b, c \in \mathbb{Z}_p$, which has a root (x_0, y_0) modulo p satisfying $|x_0| < X$ and $|y_0| < Y$. It suffices to divide by $b_{1,1}$ the polynomial from Equation (5) to make it unary in the first

variable. Lemma 3 suggests to find a small-norm polynomial $h(x, y)$ that shares its root with the initial polynomial $P(x, y)$. To do so, we construct the matrix M whose rows are formed by the coefficients of the polynomials p, pyY and $P(xX, yY)$ in the basis $(1, y, x)$.

$$M = \begin{pmatrix} p & 0 & 0 \\ 0 & pY & 0 \\ c & bY & X \end{pmatrix}$$

Using the LLL algorithm [30], we can find a small linear combination of these polynomials that will satisfy Lemma 3. Indeed, this matrix has determinant p^2XY and an LLL reduction of the basis of the lattice spanned by the rows of M will output one vector of norm upper bounded by $2^{3/4}(\det(M))^{1/3}$. We expect the second vector to behave as the first, which is confirmed experimentally.

To obtain two polynomials which satisfy Lemma 3, we need the inequality $2^{3/4}(\det(M))^{1/3} < p/\sqrt{3}$, i.e. $XY < 3^{-3/2} \cdot 2^{-9/4}p$. If $g(x, y) = g_0 + g_1x + g_2y$ and $h(x, y) = h_0 + h_1x + h_2y$ are the polynomials corresponding to the shortest vectors output by LLL, we can recover (x_0, y_0) as (if the denominator is not zero):

$$x_0 = \frac{X(h_0g_2 - g_0h_2)}{g_1h_2 - h_1g_2} \text{ and } y_0 = \frac{Y(h_0g_1 - h_1g_0)}{g_2h_1 - h_2g_1}.$$

As a consequence, this method makes it possible to recover in polynomial time any values χ and χ' that mask the secret value $a_{1,1}$ if they are both below $\sqrt{p}$. The complexity of Nguyen and Stehlé's LLL is quadratic [36], in our case it is $O(d^5 \log(3/2 \log(p))^2)$, with $d = 3$. Then $a_{1,1}$ can be computed as $a_{1,1} = c_{1,1} + b_{1,1}\chi \mod p$. (see Appendix A for a practical example of this attack). The scheme from [45] is therefore completely insecure.

Remark 2 *One could fix this issue in Wang* et al.*'s protocol by using a larger Υ and making the value χ uniformly distributed over $\mathbb{Z}_p$. This would make the protocol not more efficient for the client than the actual computation of a single exponentiation. However, even this inefficient protocol would not achieve the privacy security notion as explained in Section 8.*

4.2 Attack on Ding *et al.*'s protocol [20]

In [20], Ding, Xu, Ye and Choo propose an algorithm to delegate a modular exponentiation to a single untrusted server. Unfortunately, their method suffers the same weaknesses that Wang *et al.*'s protocol. They consider the exponentiation $u^a \mod p$, in the scenario $000v$.

During the queries to the server, the exponent a is hidden as follows:

$$a = x_1 + t_1y_1 \mod p \quad (6)$$

$$ra = x_2 + t_2y_2 \mod p, \quad (7)$$

where x_1, x_2 and y_1, y_2 are elements of $\mathbb{Z}_p$, t_1 and t_2 are "at least 64 bit long (the same as χ used in Wang *et al.* (2014)" (*sic*), and r is a small integer (it is suggested that $r \in [2, 11]$). The values x_1, x_2, y_1, y_2 are part of the queries to the server, so they are known to an adversary.

The rest of the attack consists in first guessing r (the attacker can exhaustively try them all since these values lie in a small interval), and then combining equations (6) and (7) as $r \cdot (6) - (7)$, we get

$$(y_1 r)t_1 - t_2 y_2 + x_1 r - x_2 = 0 \mod p$$

which, once again, is an equation which can be solved if t_1 and t_2 are small enough. The same analysis as before applies in this case, which means that t_1 and t_2 can be efficiently computed as soon as they are lower than $\sqrt{p}$ (which is much larger than 2^{64}). Once they are recovered, the value of a is known and the protocol is broken. The approach from [20] is therefore insecure.

5 Generic Constructions for Privately Outsourcing Exponentiation

We focus in this section on protocols for outsourcing a single exponentiation $(u, a) \mapsto u^a$. Protocols for outsourcing multi-exponentiations are given in Section 8. As mentioned in the introduction, the round complexity is the main bottleneck in improving the efficiency of secure protocols due to latency, and we consider only 1-round delegation protocols. Protocols for fixed base exponentiation are probably folklore (e.g., see [29] for a verifiable variant of the protocol corresponding to the computation code $\boldsymbol{\beta} = 001f$) but have not unpublished (to the best of our knowledge). Protocols for variable base exponentiation seem to be new and are inspired by Gallant, Lambert and Vanstone's decomposition algorithm [22].

All these protocols are secure in the (indistinguishability) privacy notion defined in Section 2 in the information-theoretic sense (see Theorem 1). Optimality results (in terms of computation in the group $\mathbb{G}$) are given in Section 6, and summed up in Table 2.

5.1 Constructions for Outsourcing Fixed Base Exponentiation

When the base u is fixed, one can assume that $\mathcal{C}$ can use a random power generator for u. As described in Section 3.4, this generator $\mathcal{B}$ is invoked with no input and outputs a single random pair $(u^k, k) \in \mathbb{G} \times \mathbb{Z}_p$ where k is uniformly distributed in $\mathbb{Z}_p$ (or statistically close to the uniform distribution). If the generator $\mathcal{B}(\cdot)$ is invoked several times, we assume that the output pairs are independent.

Trivial Cases. Obviously, the case $111f$ (everything public) is trivial (simply ask in clear to the server $\mathcal{S}$ the computation of u^a as $\mathcal{S}(u, a)$) and the case $110f$ does not make sense (public inputs and private output), as well as the case $011f$ (secret base) in the prime order setting (but the latter case may have some interest for composite order groups).

Cases where the Base is Secret ($0{*}{*}f$). If everything is secret (case $000f$), it is easy to delegate the computation of u^a for any exponent a using Protocol 1. The client computation amounts to two invocations of the generator $\mathcal{B}$, one inversion modulo p and one multiplication in $\mathbb{G}$, with only one exponentiation delegated to $\mathcal{S}$.

Even if the exponent is public (case $010f$), Protocol 1 remains the best possible in terms of multiplications in $\mathbb{G}$ (with only one invocation to $\mathcal{S}$) since there is only one multiplication and it is needed to hide the private result of the exponentiation.

If the result is public (case $001f$), one can propose the improved Protocol 2, which needs only one invocation of the random power generator, one inversion modulo p and no multiplication in $\mathbb{G}$, with only one exponentiation delegated to $\mathcal{S}$.

Cases where the Base is Public ($1**f$). If the result is public (case $101f$), Protocol 2 remains the best possible in terms of multiplications in $\mathbb{G}$ (with only one invocation to $\mathcal{S}$) since no multiplication is needed.

If the result is secret (case $100f$), Protocol 3 is the best possible in terms of multiplications in $\mathbb{G}$ since it only needs one invocation of the random power generator and one multiplication in $\mathbb{G}$ (needed to hide the private result of the exponentiation), with only one exponentiation delegated to $\mathcal{S}$.

5.2 Constructions for Outsourcing Variable Base Exponentiation

In this paragraph, we consider the case when $\mathcal{C}$ wants to delegate the computation of u^a but with a variable u. In this setting, one cannot assume that $\mathcal{C}$ can use a random power generator for u but we can still suppose that it can use a random power generator for a fixed generator g that we still call $\mathcal{B}$ with the same properties as before.

Trivial Cases. As above, the case $111v$ (everything public) is trivial (simply ask in clear to the server $\mathcal{S}$ the computation of u^a as $\mathcal{S}(u, a)$) and the case $110v$ does not make sense (public inputs and private output), as well as the case $011v$ (secret base) in the prime order setting.

Cases where the Base is Public ($1**v$). We first consider the case where the variable base u can be made public but not the exponent nor the result (case $100v$). We propose a family of protocols depending on a parameter s that perform the computation of u^a by delegating s exponentiations to a server and doing $\log(p)/(s+1)$ operations in $\mathbb{G}$.

This family of protocols is given in Protocol 5 and the specific case $s = 1$ is Protocol 4. Note that these protocols do not make use of the random power generator for g. Unfortunately, the efficiency gain is only a factor s and if the number of delegated exponentiations is constant the client still has to perform $O(\log p)$ operations in $\mathbb{G}$.

These protocols are actually optimal in terms of operations in $\mathbb{G}$, as we show in Theorems 2 and 3. Obviously, we can also use these protocols if we allow the result u^a to be public (case $101v$) and the optimal result of Theorems 2 and 3 show that even in this easier setting, the protocol cannot be improved.

Cases where the Base is Private ($0**v$). We can use this protocol family to construct another delegation protocol for the corresponding cases where the base is kept secret ($000v$ and $001v$). We obtain Protocol 6 that makes two invocations

Table 2 Outsourcing protocols for single exponentiation

Constructions for Outsourcing Fixed Base Exponentiation (with a random power generator of pairs (k, u^k) available)									
Code	u	a	u^a	Resources	Secure constructions		Complexity Lower Bound (for $\mathcal{G}$)		Optimality
					Protocol	Complexity	Lower Bound	Proof	
111f	Public	Public	Public	Trivial					
110f	Public	Public	Secret	Non-sense					
101f	Public	Secret	Public	$(1\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 2	$1\ \mathcal{S} \quad + 1\ \mathcal{B}$	$0\ \mathcal{G}$	From Protocol 2	✓
100f	Public	Secret	Secret	$(1\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 3	$1\ \mathcal{S} + \quad 1\ \mathcal{G} + 1\ \mathcal{B}$	$1\ \mathcal{G}$	From Protocol 3	✓
011f	Secret	Public	Public	Non-sense†					
010f	Secret	Public	Secret	$(1\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 1	$1\ \mathcal{S} + \quad 1\ \mathcal{G} + 2\ \mathcal{B}$	$1\ \mathcal{G}$	From Protocol 1	✓
001f	Secret	Secret	Public	$(1\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 2	$1\ \mathcal{S} \quad + 1\ \mathcal{B}$	$0\ \mathcal{G}$	From Protocol 2	✓
000f	Secret	Secret	Secret	$(1\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 1	$1\ \mathcal{S} + \quad 1\ \mathcal{G} + 2\ \mathcal{B}$	$1\ \mathcal{G}$	From Case $010f$	✓

Constructions for Outsourcing Variable Base Exponentiation (with a random power generator of pairs (k, g^k) available)									
Code	u	a	u^a	Resources	Secure constructions		Complexity Lower Bound (for $\mathcal{G}$)		Optimality
					Protocol	Complexity	Lower Bound	Proof	
111v	Public	Public	Public	Trivial					
110v	Public	Public	Secret	Non-sense					
101v	Public	Secret	Public	$(1\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 4	$1\ \mathcal{S} + \quad L_p/2\ \mathcal{G}$	$L_p/2\ \mathcal{G}$	Heuristic (Rem. 7)	?**
				$(s\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 5	$s\ \mathcal{S} + L_p/(s+1)\ \mathcal{G}$	$L_p/(s+1)\ \mathcal{G}$	Heuristic (Rem. 7)	?**
100v	Public	Secret	Secret	$(1\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 4	$1\ \mathcal{S} + \quad L_p/2\ \mathcal{G}$	$L_p/2\ \mathcal{G}$	Th. 2, Sec. 6	✓
				$(s\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 5	$s\ \mathcal{S} + L_p/(s+1)\ \mathcal{G}$	$L_p/(s+1)\ \mathcal{G}$	Th. 3, Sec. 6	✓
011v	Secret	Public	Public	Non-sense†					
010v	Secret	Public	Secret	$(1\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 8	$1\ \mathcal{S} + \quad L_p/\ell\ \mathcal{G} + \ell\ \mathcal{B}$	$L_p/(2\ell+4)\ \mathcal{G}$	Th. 4, Sec. 6	?
				$(2\ \mathcal{S}, \ell\ \mathcal{B})$	Protocol 7	$2\ \mathcal{S} + \quad 3\ \mathcal{G} + 3\ \mathcal{B}$	$t^{\ddagger}\ \mathcal{G}$	From Protocol 7	?
001v	Secret	Secret	Public	$(1\ \mathcal{S}, \ell\ \mathcal{B})$			$L_p/2\ \mathcal{G}$	From Case $101v$	?
				$(2\ \mathcal{S}, \ell\ \mathcal{B})$	6 (using 4)	$2\ \mathcal{S} + \quad L_p/2\ \mathcal{G} + 2\ \mathcal{B}$	$L_p/3\ \mathcal{G}$	From Case $101v$	?
				$(s\ \mathcal{S}, \ell\ \mathcal{B})$	6 (using 5)	$s\ \mathcal{S} + \quad L_p/s\ \mathcal{G} + 2\ \mathcal{B}$	$L_p/(s+1)\ \mathcal{G}$	From Case $101v$	?
000v	Secret	Secret	Secret	$(1\ \mathcal{S}, \ell\ \mathcal{B})$			$L_p/2\ \mathcal{G}$	From Case $100v$	?
				$(2\ \mathcal{S}, \ell\ \mathcal{B})$	6 (using 4)	$2\ \mathcal{S} + \quad L_p/2\ \mathcal{G} + 2\ \mathcal{B}$	$L_p/3\ \mathcal{G}$	From Case $100v$	?
				$(s\ \mathcal{S}, \ell\ \mathcal{B})$	6 (using 5)	$s\ \mathcal{S} + \quad L_p/s\ \mathcal{G} + 2\ \mathcal{B}$	$L_p/(s+1)\ \mathcal{G}$	From Case $100v$	?

Notations: $\ell = O(1)$ and $L_p = \log(p)$. † Prime order setting. ‡ With $t \in \{0, 1, 2, 3\}$. ** refers to Remark 7.

Protocol 1: $000f$ (and $010f$)

Input: $u \in \mathbb{G}$, $a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$
$(u^r, r) \leftarrow \mathcal{B}(\cdot)$
$(u^s, s) \leftarrow \mathcal{B}(\cdot)$
$t \leftarrow (a - s)/r \bmod p$
$h \leftarrow \mathcal{S}(u^r, t \bmod p)$
return $h \cdot u^s$

Protocol 2: $001f$ (and $101f$)

Input: $u \in \mathbb{G}$, $a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$
$(u^k, k) \leftarrow \mathcal{B}(\cdot)$
$h \leftarrow \mathcal{S}(u^k, a/k \bmod p)$
return h

Protocol 3: $100f$

Input: $u \in \mathbb{G}$, $a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$
$(u^k, k) \leftarrow \mathcal{B}(\cdot)$
$h \leftarrow \mathcal{S}(u, a - k \bmod p)$
return $h \cdot g^k$

Fig. 3 Delegation protocols for fixed base exponentiation

of the random generator for g and requires the delegation of one further exponentiation compared to Protocol 5 (and Protocol 4). We do not actually know if these protocols are optimal but the gap is rather tight (see Table 2).

Constructing an outsourcing protocol in these cases with only one exponentiation delegation (or proving it is impossible) is left as an open problem.

We can also use this protocol if we allow the exponent a to be public ($010v$). However, in this case one can improve it with Protocol 7 where the client performs only a constant number of group operations in $\mathbb{G}$. In this case, one can also improve it with Protocol 8 where the client makes only one call to the server, but at the price of a $O(\log(p))$ number of group operations in $\mathbb{G}$.

Remark 3 *In [11], Cavallo* et al. *presented two other protocols for outsourcing private variable base and public exponent exponentiation. The first one [11, §4, p. 164], recalled in Protocol 9, achieves only the basic security requirement (i.e., in the sense of one-wayness instead of indistinguishability). It relies on a subset-sum in a group and in order to achieve a stronger privacy notion, the delegation scheme actually becomes less efficient for the client than performing the exponentiation on its own. The second scheme is much more efficient since the client computation is constant but it requires a stronger random power generator $\mathcal{B}$ that outputs random triples of the form (g^r, g^{ar}, r). In particular, this second protocol can only be used for fixed values of the public exponent a.*

Theorem 1 *Let* $\mathsf{GroupGen}$ *be a group generator, let λ be a security parameter and let $\mathbb{G}$ be a group of prime order p output by* $\mathsf{GroupGen}(\lambda)$*. Let $(\mathcal{C}, \mathcal{S})$ be one client-server protocol for the delegated computation of the exponentiation u^a described in Protocols 1 – 8 (for the corresponding computation code β given in their description). The protocol $(\mathcal{C}, \mathcal{S})$ satisfies unconditionally $(\tau, 0)$-indistinguishability against a malicious adversary for any time τ.*

Protocol 4: $100v$ (and $101v$)

Input: $u \in \mathbb{G}$, $a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$
  $T \leftarrow \lceil \sqrt{p} \rceil$
  $h \leftarrow \mathcal{S}(u, T)$
  $a_0 \leftarrow a \bmod T$; $a_1 \leftarrow a \operatorname{div} T$ ▷ Euclidean division: $a = a_1 \cdot T + a_0$
  return $u^{a_0} h^{a_1}$ ▷ using Algorithm 1

Protocol 5: $100v$ (and $101v$)

Input: $u \in \mathbb{G}$, $a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$
  $T \leftarrow \lceil p^{1/s+1} \rceil$
  for i from 1 to s **do**
    $h_i \leftarrow \mathcal{S}(u, T^i)$
  end for
  $\mathsf{temp} \leftarrow a$
  for i from s down to 0 **do**
    $a_i \leftarrow \mathsf{temp} \operatorname{div} T^i$ ▷ $a = a_s \cdot T^s + \cdots + a_1 T + a_0$
    $\mathsf{temp} \leftarrow \mathsf{temp} - a_i \cdot T^i$
  end for
  return $u^{a_0} \prod_{i=1}^{s} h_i^{a_i}$ ▷ using Algorithm 1

Protocol 6: $000v$ (and $001v$)

Input: $u \in \mathbb{G}$, $a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$
  $(g^{k_1}, k_1) \leftarrow \mathcal{B}(\cdot)$
  $v \leftarrow u \cdot g^{k_1}$
  $h_1 \leftarrow v^a$ ▷ delegated with Protocol 4 or 5 (public base); $h_1 = v^a = u^a \cdot g^{ak_1}$
  $(g^{k_2}, k_2) \leftarrow \mathcal{B}(\cdot)$
  $h_2 \leftarrow \mathcal{S}(g, -ak_1 - k_2 \bmod p)$ ▷ $h_2 = g^{-ak_1 - k_2}$
  return $h_1 \cdot h_2 \cdot g^{k_2}$

Protocol 7: $010v$

Input: $u \in \mathbb{G}$, $a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$
  $(g^r, r) \leftarrow \mathcal{B}(\cdot)$; $(g^s, s) \leftarrow \mathcal{B}(\cdot)$; $(g^t, t) \leftarrow \mathcal{B}(\cdot)$
  $k \leftarrow (t - ra)/s \bmod p$
  $h_1 \leftarrow \mathcal{S}(u \cdot g^r, a)$
  $h_2 \leftarrow \mathcal{S}(g^s, k)$
  return $h_1 h_2 g^t$

Protocol 8: $010v$

Input: $u \in \mathbb{G}$, $a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$
  $(g^r, r) \leftarrow \mathcal{B}(\cdot)$
  for i from 1 to s **do**
    $(g^{t_i}, t_i) \leftarrow \mathcal{B}(\cdot)$
  end for
  $(k_0, k_1, \ldots, k_s) \leftarrow \text{GLV-Dec}(1, t_1, \ldots, t_s, -ra \bmod p)$ ▷ with $k_i \leq p^{1/(s+1)}$
  $h_1 \leftarrow \mathcal{S}(u \cdot g^r, a)$
  $h_2 \leftarrow g^{k_0} (g^{t_1})^{k_1} \ldots (g^{t_s})^{k_s}$ ▷ using Algorithm 1
  return $h_1 h_2$

Protocol 9: $010v$ from [11]

Input: $u \in \mathbb{G}$, $a \in \mathbb{Z}_p$
Output: $u^a \in \mathbb{G}$
  for i from 1 to s **do**
    $g_i \xleftarrow{R} \mathbb{G}$
  end for
  $\mathcal{I} \xleftarrow{R} \mathfrak{P}_m(\{1, \ldots, s\})$ ▷ random subset of cardinal m of $\{1, \ldots, s\}$
  $g_{s+1} \leftarrow u \cdot \prod_{i \in \mathcal{I}} g_i$
  for i from 1 to s **do**
    $h_i \leftarrow \mathcal{S}(g_i, -a)$
  end for
  $h_{s+1} \leftarrow \mathcal{S}(g_{s+1}, a)$
  return $h_{s+1} \cdot \prod_{i \in \mathcal{I}} h_i$

Fig. 4 Delegation protocols for variable base exponentiation

Proof Since the protocols (and thus the proofs) are all very similar, we focus on Protocol 1. The correctness follows from the equality

$$h \cdot u^s = [(u^r)^{t \bmod p}]u^s = [(u^r)^{(a-s)/r \bmod p}]u^s = u^a.$$

We now prove that there is no adversary $\mathcal{A}$ (running in any time τ) for the privacy security notion from Definition 3. The adversary chooses a group element u and two scalars $(a_0, a_1) \in \mathbb{Z}_p^2$. The challenger picks uniformly at random a bit $b \in \{0,1\}$ and sets $a = a_b$. The client runs the delegation protocol with inputs u and a and delegates one exponentiation to the adversary acting as the server. The adversary has to guess the bit b.

Due to the properties of the random power generator, r and s are uniformly distributed in $\mathbb{Z}_p$, so that t is also uniformly distributed in $\mathbb{Z}_p$ and does not depend on the value a. The invocation $\mathcal{S}(u^r, t \bmod p)$ thus does not reveal anything on the value a (in an information-theoretic sense), meaning that the advantage of the adversary in guessing the bit b is 0.

Remark 4 *Theorem 1 asserts that our protocols achieves unconditionally the privacy experiment described in Fig. 2 (i.e., in the information theoretic sense). In the two cases $\beta = 101v$ and $\beta = 101f$, the adversary is given the input base $\mathbf{u}$ and the result $\mathbf{u}^{\mathbf{a}}$. In the special case $n = 1$, Theorem 1 is trivially true (since there is only one possible exponent $\mathbf{a} = a \in \mathbb{Z}_p$ and the predicate $P_2((\mathbf{a}_0^\star, \mathbf{u}_0^\star), (\mathbf{a}_1^\star, \mathbf{u}_1^\star), \boldsymbol{\beta})$ is satisfied only if $\mathbf{a}_0^\star = \mathbf{a}_1^\star$. For $n > 1$, Theorem 1 asserts that an adversary learns no information (in an information theoretic sense) on the representation of $\mathbf{u}^{\mathbf{a}}$ in base $\mathbf{u}$ from the protocol execution.*

Remark 5 *In this paper, we do not consider the setting where the client can also store precomputed values (in addition to having access to a random power generator $\mathcal{B}$ and a server oracle $\mathcal{S}$). For instance, in Protocol 8, we use the random power generator s times in order to generate pairs (g^{t_i}, t_i) for $i \in \{1, \ldots, s\}$ and then the GLV decomposition algorithm (*GLV-Dec*) in order to decompose the scalar $-ra \bmod p$ as $-ra = k_0 + k_1 t_1 + \cdots + k_s t_s \bmod p$ with "small" scalars $k_i \leq p^{1/(s+1)}$. Actually, the pairs (g^{t_i}, t_i) for $i \in \{1, \ldots, s\}$ can be re-used and they do not need to be random to ensure privacy. We can thus consider a simpler variant of Protocol 8 in which the client stores precomputed values g^{t_i} with $t_i = T^i$ for $T = \lceil p^{1/(s+1)} \rceil$ for $i \in \{0, \ldots, s\}$ and decomposes the scalar $-ra \bmod p$ in base T as $-ra = k_0 t_0 + k_1 t_1 + \cdots + k_s t_s \bmod p$ with "small" scalars $k_i < T$ as in Protocol 5. The resulting protocol is then simpler and more efficient in practice. However, it has the same "oracle complexity" since it only replaces queries to the random power generator $\mathcal{B}$ by storage of precomputed values. It can be easily seen that the complexity lower bounds from Section 6 can be generalized to this setting.*

6 Complexity Lower Bound for One-Round Protocols

We focus on studying protocols with minimal interaction, namely the client is allowed to delegate the computation of several group exponentiations but it must send all of them to the server in only one communication round. Indeed, interactions over computer networks are usually the most time consuming operations (due to lagging or network congestion) and it is very important to study protocols

which require the minimal number of rounds to complete. In Section 7, we also present complexity lower bounds for multi-round protocols.

By "lower bounds", we mean that the number of calls to the server oracle $\mathcal{S}$ and to the random power generator $\mathcal{B}$ are fixed, and that we consider the number of group operations. All the results concerning this section are summed up in the column "Complexity Lower Bound" of Table 2. The last column of Table 2 gives a hint for the proof of those lower bounds. Concerning the first part of the table, the bounds come from the protocols given in Section 5, since at least one call to the group oracle is mandatory when the result is private (the client $\mathcal{C}$ needs to do at least one computation after having received a public result from the server oracle $\mathcal{S}$). The cases $101v$ and $100v$ are then dealt with in Theorem 3. For all these cases, the protocols proposed in Section 5 are thus actually optimal. As for Case $010v$, the lower bound for a unique call to $\mathcal{S}$ is proven in Theorem 4, whereas Protocol 7 gives a (constant) upper bound in case we allow a second call to $\mathcal{S}$. Finally, the lower bounds for Cases $001v$ and $000v$ come from the equivalent bounds for Cases $101v$ and $100v$, since the variable base is furthermore assumed to be secret.

In what follows, and as mentioned above, we use the generic group model to prove these lower bounds. We model the different operations as follows:

- The group oracle $\mathcal{G}$ takes as inputs two encodings $\sigma_1 = \sigma(h_1)$ and $\sigma_2 = \sigma(h_2)$ and outputs the encoding σ_3 such $\sigma_3 = \sigma(h_1 h_2)$ (see Section 3.1).
- The random power generator $\mathcal{B}$ outputs pairs $(t, \sigma(g^t))$ where the scalar t is picked uniformly at random in $\mathbb{Z}_p^*$ (independently for all queries).
- The server oracle $\mathcal{S}$ takes as inputs an encoding $\sigma_0 = \sigma(h)$ and a scalar x and outputs the encoding $\sigma_0' = \sigma(h^x)$ (*i.e.* $\sigma^{-1}(\sigma_0') = \sigma^{-1}(\sigma_0)^x$).

The following theorems assert that for the case $100v$, the protocols proposed in Section 5 are actually optimal in terms of calls to $\mathcal{S}$ and $\mathcal{G}$.

For the ease of exposition, we first state our result and present a proof for the simple case where the client $\mathcal{C}$ outsources only one exponentiation to the server $\mathcal{S}$:

Theorem 2 *Let* GroupGen *be a group generator and let* $(\mathcal{C}, \mathcal{S})$ *be one client-server protocol for the delegated computation of the exponentiation* u^a *for the corresponding computation code* $\boldsymbol{\beta} = 100v$*. We assume that the client* $\mathcal{C}$ *is a generic group algorithm that uses*

- $c \log(p) + O(1)$ *generic group operations (for all groups* $\mathbb{G}$ *of prime order* p *output by* $\mathsf{GroupGen}(\lambda)$*) for some constant* c*,*
- $\ell = O(1)$ *queries to the (private) random power generator* $\mathcal{B}$
- *and only* 1 *delegated exponentiation to the server* $\mathcal{S}$

If $c < 1/2$*, then* $(\mathcal{C}, \mathcal{S})$ *is not indistinguishable: there exists an honest-but-curious algorithm running in polynomial-time and a constant* $\kappa > 0$ *such that*

$$\Pr[\nu = 1 | \nu \leftarrow \mathbf{Exp}_{ind}(\mathcal{A}, \boldsymbol{\beta}, \lambda)] \geq 1 - \lambda^{-\kappa}.$$

Proof We assume that $\mathcal{C}$ gets as input two encodings $\sigma(u)$, $\sigma(g)$ of two group elements u and g and one scalar a in $\mathbb{Z}_p$ and outputs the encoding $\sigma(u^a)$ of the group element u^a by making q queries to the group oracle $\mathcal{G}$, ℓ queries to the (private) random power generator $\mathcal{B}$ and 1 query to $\mathcal{S}$.

We assume that $q = c \log p + O(1)$ with $c < 1/2$ and we prove that it is not possible for $\mathcal{C}$ to compute $\sigma(u^a)$ in such a way that the server $\mathcal{S}$ learns no information on a. More precisely, we construct a polynomial-time adversary $\mathcal{A}$ for the privacy security notion from Definition 3. The adversary chooses a group element u and two scalars $(a_0, a_1) \in \mathbb{Z}_p^2$. For the sake of simplicity, we assume that the adversary picks $(a_0, a_1) \in \mathbb{Z}_p^2$ uniformly at random among the scalars of bit-length $\log(p)$ and u uniformly at random in $\mathbb{G}$. The challenger picks uniformly at random a bit $b \in \{0, 1\}$ and sets $a = a_b$. The client runs the delegation protocol with inputs u and a and delegates one exponentiation to the adversary acting as the server. The adversary has to guess the bit b.

Let us denote $(t_1, \sigma(g^{t_1}))$, $(t_2, \sigma(g^{t_2}))$, $\ldots$, $(t_\ell, \sigma(g^{t_\ell}))$ the pairs obtained from the random power generator $\mathcal{B}$ by the client $\mathcal{C}$. Since $\mathcal{B}$ takes no inputs and outputs independent pairs, we can assume without loss of generality that the client $\mathcal{C}$ makes the ℓ queries to $\mathcal{B}$ in a first phase of the delegation protocol. We denote $(\sigma(h), x)$ the unique pair encoding of group element/scalar made by $\mathcal{C}$ to the server $\mathcal{S}$ (which is executed by the adversary $\mathcal{A}$ in an "honest-but-curious" way). Using generic group operations, $\mathcal{C}$ can only construct the corresponding group element such that:

$$h = u^{\alpha'} \cdot g^{\kappa'} \cdot g^{t_1 \gamma'_1} \cdots g^{t_\ell \gamma'_\ell} \tag{8}$$

for some scalars $(\alpha', \kappa', \gamma'_1, \ldots, \gamma'_\ell)$. We denote $k = h^x$ the response of $\mathcal{S}$. Eventually, the client $\mathcal{C}$ outputs the encoding $\sigma(u^a)$ of the group element u^a. Again, using generic group operations, it can only construct it as

$$u^a = u^\alpha g^\kappa \cdot g^{t_1 \gamma_1} \cdots g^{t_\ell \gamma_\ell} k^\delta \tag{9}$$

for some scalars $(\alpha, \kappa, \gamma_1, \ldots, \gamma_\ell, \delta)$. If we assume that $q = c \log p + O(1)$ (and in particular $q = o(\sqrt{p})$), the client $\mathcal{C}$ is not able to compute the discrete logarithm of u in base g. This means that necessarily the exponent of u and g in Equation (9) cancels out with overwhelming probability and we obtain

$$a = \alpha + \delta \alpha' x \mod p. \tag{10}$$

.

Indeed, if this is not the case, then the client $\mathcal{C}$ can be transformed readily into a generic algorithm which solves the discrete logarithm problem in $\mathbb{G}$ with the same computational complexity. Equations (9) and (8) leads to the relation

$$u^{a - \alpha - \alpha' \delta x} = g^{(\kappa + x\delta\kappa') + (t_1 + x\delta t'_1)\gamma_1 + (t_\ell + x\delta t'_\ell)\gamma_\ell}$$

and if the exponents do not cancel out, one can compute the discrete logarithm of u in base g as

$$\frac{(\kappa + x\delta\kappa') + (t_1 + x\delta t'_1)\gamma_1 + (t_\ell + x\delta t'_\ell)\gamma_\ell}{a - \alpha - \alpha'\delta x} \mod p$$

The query complexity of this generic algorithm is $q = c \log p + O(1)$ and Shoup's result on the generic complexity of the discrete logarithm problem [42, Theorem 1] implies that this happens with probability at most $O(q^2/p) = O(\log^2 p/p)$.

We denote τ_1 the number of group operations performed by $\mathcal{C}$ in the computation of h described in Equation (8) and τ_2 the additional number of operations

in the computation of u^a described in Equation (9) (after receiving k). By assumption, $\tau_1 + \tau_2 \leq c \log p + O(1)$. Furthermore, since $\mathcal{C}$ only used generic group operations, we have (by Lemma 2 in Section 3.1) $\alpha \leq 2^\tau$, $\alpha' \leq 2^{\tau_1}$, and $\delta \leq 2^{\tau_2}$. If we note $\rho_1 = \alpha$ and $\rho_2 = \delta\alpha'$, Equation (10) becomes $a = \rho_1 + x\rho_2 \mod p$, where x is known to the adversary, $\rho_2 = \delta\alpha' \leq 2^{\tau_1} 2^{\tau_2} = 2^{\tau_1+\tau_2} \leq 2^\tau \leq p^{c+o(1)}$ and $\rho_1 = \alpha \leq 2^\tau \leq p^{c+o(1)}$.

The adversary $\mathcal{A}$ can then try to decompose a_0 and a_1 as $a_i = \rho_{i,1} + x\rho_{i,2} \mod p$, with $\rho_{i,1}, \rho_{i,2} \leq p^{c+o(1)}$. For $a_b = a$, the decomposition algorithm from Section 3.3 (which generalizes the main attack on Wang *et al.* or Ding *et al.*'s protocols) will recover $\rho_{b,1}$ and $\rho_{b,2}$ in polynomial time. However, for a given x and a random a_{1-b} of bit-length $\log(p)$, there is only a negligible probability that such a decomposition exists (less than $p^{c+o(1)} \times p^{c+o(1)} = p^{2c+o(1)} = o(p)$ scalars can be written in this way). Thus, the adversary can simply run the decomposition algorithm from Section 3.3 on (a_0, x) on one hand and on (a_1, x) on the other hand and returns the bit b for which the algorithm returns a "small decomposition" on input (a_b, x). By the previous analysis, its advantage is noticeable.

Theorem 3 generalizes Theorem 2 and consider the general case where the client $\mathcal{C}$ outsources $s \geq 1$ exponentiations to the server $\mathcal{S}$:

Theorem 3 *Let* $\mathsf{GroupGen}$ *be a group generator and let* $(\mathcal{C}, \mathcal{S})$ *be one client-server protocol for the delegated computation of one exponentiation for the computation code* $\boldsymbol{\beta} = 100v$*. We assume that the client* $\mathcal{C}$ *is a generic group algorithm that uses*

- $c \log(p) + O(1)$ *generic group operations (for groups* $\mathbb{G}$ *of prime order* p *output by* $\mathsf{GroupGen}(\lambda)$*),*
- $\ell = O(1)$ *queries to the (private) random power generator* $\mathcal{B}$
- *and* s *simultaneous delegated exponentiation to the server* $\mathcal{S}$

If c *satisfies* $c < 1/(s+1)$*, then* $(\mathcal{C}, \mathcal{S})$ *is not indistinguishable: there exists an honest-but-curious algorithm running in polynomial-time and a constant* $\kappa > 0$ *such that*

$$\Pr[\nu = 1 | \nu \leftarrow \mathbf{Exp}_{ind}(\mathcal{A}, \boldsymbol{\beta}, \lambda)] \geq 1 - \lambda^{-\kappa}.$$

Proof We assume that the client $\mathcal{C}$ gets as input two encodings $\sigma(u)$, $\sigma(g)$ of *two* group elements u and g picked uniformly at random in $\mathbb{G}$ and one scalar a picked uniformly at random in $\mathbb{Z}_p$ and outputs the encoding $\sigma(u^a)$ of the group element u^a by making only

- q queries to the group oracle $\mathcal{G}$;
- ℓ queries to the random power generator $\mathcal{B}$;
- s simultaneous queries to the server oracle $\mathcal{S}$.

We assume that $q = c \log p + O(1)$ with $c < 1/(s+1)$ and we prove that it is not possible for $\mathcal{C}$ to compute $\sigma(u^a)$ in such a way that the server $\mathcal{S}$ learns no information on a.

More precisely, we construct a polynomial-time adversary $\mathcal{A}$ for the privacy security notion from Definition 3. The adversary chooses a group element $u \in \mathbb{G}$ and two scalars $(a_0, a_1) \in \mathbb{Z}_p^2$. As above, for the sake of simplicity, we assume that the adversary picks $(a_0, a_1) \in \mathbb{Z}_p^2$ uniformly at random among the scalars of bit-length $\log(p)$ and u uniformly at random in $\mathbb{G}$. The challenger picks uniformly at random a bit $b \in \{0, 1\}$ and sets $a = a_b$. The client runs the delegation protocol

with inputs u and a and delegates s exponentiations to the adversary acting as the server. The adversary has to guess the bit b.

Let us denote $(t_1, \sigma(g^{t_1}))$, $(t_2, \sigma(g^{t_2}))$, $\ldots$, $(t_\ell, \sigma(g^{t_\ell}))$ the pairs obtained from the random power generator $\mathcal{B}$ by the client $\mathcal{C}$. Since the random power generator $\mathcal{B}$ takes no inputs and outputs independent pairs, we can assume without loss of generality that the client $\mathcal{C}$ makes the ℓ queries to $\mathcal{B}$ in a first phase of the delegation protocol.

We denote $(\sigma(h_1), x_1)$, $\ldots$, $(\sigma(h_s), x_s)$ the pairs group element/scalar made by $\mathcal{C}$ to the server $\mathcal{S}$ (which is executed by the adversary $\mathcal{A}$ in an "honest-but-curious" way). Using generic group operations, $\mathcal{C}$ can only construct the corresponding group elements such that:

$$\begin{aligned} h_1 &= u^{\alpha_1} \cdot g^{\kappa_1} \cdot g^{t_1\gamma_{1,1}} \cdots g^{t_\ell\gamma_{1,\ell}} \\ h_2 &= u^{\alpha_2} \cdot g^{\kappa_2} \cdot g^{t_1\gamma_{2,1}} \cdots g^{t_\ell\gamma_{2,\ell}} \\ h_3 &= u^{\alpha_3} \cdot g^{\kappa_3} \cdot g^{t_1\gamma_{3,1}} \cdots g^{t_\ell\gamma_{3,\ell}} \\ &\vdots \\ h_s &= u^{\alpha_s} \cdot g^{\kappa_s} \cdot g^{t_1\gamma_{s,1}} \cdots g^{t_\ell\gamma_{s,\ell}} \end{aligned} \tag{11}$$

for some scalars $(\alpha_1, \ldots, \alpha_s)$, $(\kappa_1, \ldots, \kappa_s)$ and $(\gamma_{i,j})_{i=1,s;j=1,\ell}$. We note $k_i = h_i^{x_i}$ the response of the server $\mathcal{S}$ to the i-th query. Eventually, the client $\mathcal{C}$ outputs the encoding $\sigma(u^a)$ of the group element u^a and as above using generic group operations, it can only construct it as

$$u^a = u^\alpha g^\kappa \cdot g^{t_1\gamma_1} \cdots g^{t_\ell\gamma_\ell} k_1^{\delta_1} k_2^{\delta_2} \ldots k_s^{\delta_s} \tag{12}$$

for some scalars $(\alpha, \kappa, \gamma_1, \ldots, \gamma_\ell, \delta_1, \ldots, \delta_s)$.

As above, if we assume that $q = c \log n + O(1)$ (and in particular $q = o(\sqrt{p})$), the client $\mathcal{C}$ is not able to compute the discrete logarithm of u in base g. This means that necessarily the exponents of g and u in Equation (12) cancel out. Recall that $k_i = h^{x_i}$ for all index i, h_i being constructed as in Equation (11). Thus, taking only the discrete logarithms of powers of u in base u of this equation, we obtain

$$a = \alpha + \sum_{i=1}^{s} \delta_i \alpha_i x_i \mod p. \tag{13}$$

We denote τ_1 the number of group operations performed by $\mathcal{C}$ in the computation of $(h_1, \ldots, h_s)$ described in Equation (11) and τ_2 the additional number of group operations performed by $\mathcal{C}$ in the computation of u^a described in Equation (12) (after receiving $(k_1, \ldots, k_s)$).

By assumption, $\tau_1 + \tau_2 \leq \tau \leq c \log p + O(1)$. Furthermore, since $\mathcal{C}$ only used generic group operations, we have (by Lemma 2) $\alpha_i \leq 2^{\tau_1}$, $\alpha \leq 2^\tau$ and $\delta_i \leq 2^{\tau_2}$ for $i \in \{1, \ldots, s\}$. If we note $\mu_0 = \alpha$ and $\mu_i = \delta_i \alpha_i$ for $i \in \{1, \ldots, s\}$, Equation (13) becomes

$$a = \mu_0 + \mu_1 x_1 + \mu_2 x_2 + \mu_3 x_3 + \cdots + \mu_s x_s \mod p \tag{14}$$

where x is known to the adversary, $\mu_i = \delta_i \alpha_i \leq 2^{\tau_2} 2^{\tau_1} \leq p^{c+o(1)}$ for $i \in \{1, \ldots, s\}$ and $\mu_0 = \alpha \leq \tau \leq p^{c+o(1)}$.

Therefore the server $\mathcal{S}$ knows that a satisfies the equation (14), in which it knows the value $x_1, \ldots, x_s$ and we have $\mu_i = o(p^{1/s})$. The adversary can then try to decompose a_0 and a_1 as

$$a_b = \mu_{b,0} + \mu_{b,1}x_1 + \mu_{b,2}x_2 + \mu_{b,3}x_3 + \cdots + \mu_{b,s}x_s \mod p$$

with $\mu_{0,i}, \mu_{1,i} \leq p^{c+o(1)}$ for $i \in \{0, \ldots, s\}$. For $a_b = a$, the by using the decomposition algorithm from Section 3.3 will recover the values $\mu_{i,b}$ in polynomial time (or a potentially even shorter decomposition). Once again, for a given x and a random $a_{1-b} = a^*$ of bit-length $\log(p)$, there is only a negligible probability that such a decomposition exists (less than $(p^{c+o(1)})^{s+1} = p^{(s+1)c+o(1)} = o(p)$ scalars a can be written in this way). Thus, the adversary can simply run the decomposition algorithm from Section 3.3 on (a_0, x) on one hand and on (a_1, x) on the other hand and returns the bit b for which the algorithm returns a "small decomposition" on input (a_b, x). By the previous analysis, its advantage is noticeable.

Remark 6 *It is worth mentioning that even in (generic) groups where division is significantly less expensive than multiplication (such as elliptic curves or class groups of imaginary quadratic number fields), this lower bound (as well as the following ones) still holds (see Appendix B for details).*

Remark 7 *As mentioned above, for the case* $101v$ *(when* $n = 1$*), the privacy notion is trivially achieved by any protocol (even if the client sends directly the couple* (u, a) *to the server). It is therefore impossible to prove a lower bound for delegation protocols that achieved privacy for the case* $101v$*. However, following the proof of Theorems 2 and 3, one can argue (heuristically) that any generic protocol for the case* $101v$ *with the same query complexities (to the group oracle, to the (private) random power generator* $\mathcal{B}$ *and to the server* $\mathcal{S}$*) cannot achieve instance-hiding (assuming the discrete logarithm assumption in* $\mathsf{GroupGen}$*). The proof is identical except that in the final step, the decomposition algorithm may output a "small decomposition" which does not permit to retrieve the actual exponent* a *(even if this is very unlikely). It is an interesting open problem to remove this obstruction in the proof and to formally prove that our Protocols 4 and 5, for* $n = 1$ *are actually optimal among instance-hiding protocols (which we conjecture to be true).*

Protocol 7 shows that it is possible to delegate a secret base, public exponent exponentiation with only a constant number of operations if the client can delegate at least two exponentiations. Theorem 4 asserts that if the client is only allowed to delegate one exponentiation then Protocol 8 is almost optimal in this setting. More precisely, we show that the client has to perform at least $\Omega(\log(p))$ group operations if it delegates only one exponentiation and makes at most a constant number of queries to the random power generator $\mathcal{B}$.

Theorem 4 *Let* $\mathsf{GroupGen}$ *be a group generator and let* $(\mathcal{C}, \mathcal{S})$ *be one client-server protocol for the delegated computation of one exponentiation for the computation code* $\boldsymbol{\beta} = 010v$*. We assume that the client* $\mathcal{C}$ *is a generic group algorithm that uses*

- $c\log(p) + O(1)$ *generic group operations (for groups* $\mathbb{G}$ *of prime order* p *output by* $\mathsf{GroupGen}(\lambda)$*),*
- $\ell = O(1)$ *queries to the (private) random power generator* $\mathcal{B}$
- *and only* 1 *delegated exponentiation to the server* $\mathcal{S}$

If the constant c satisfies $c < 1/(2\ell + 4)$, then $(\mathcal{C}, \mathcal{S})$ is not indistinguishable: there exists an honest-but-curious algorithm running in time $O(p^{c/2+o(1)})$ such that

$$\Pr[\nu = 1 | \nu \leftarrow \mathbf{Exp}_{ind}(\mathcal{A}, \boldsymbol{\beta}, \lambda)] \geq 1 - \lambda^{-\kappa}.$$

Proof We assume that the client $\mathcal{C}$ gets as input two encodings $\sigma(u)$, $\sigma(g)$ of *two* group elements u and g picked uniformly at random in $\mathbb{G}$ and one scalar a picked uniformly at random in $\mathbb{Z}_p$ and outputs the encoding $\sigma(u^a)$ of the group element u^a by making only

- q queries to the group oracle $\mathcal{G}$;
- ℓ queries to the random power generator $\mathcal{B}$;
- 1 query to the server oracle $\mathcal{S}$.

We assume that ℓ is constant (with respect to the underlying group order). We assume that $q = c \log p + O(1)$ with $c < 1/(2\ell+4)$ and we prove that it is not possible for $\mathcal{C}$ to compute $\sigma(u^a)$ in such a way that the server $\mathcal{S}$ learns no information on u.

More precisely, we construct a polynomial-time adversary $\mathcal{A}$ for the privacy security notion from Definition 3. The adversary chooses two scalars $(u_0, u_1) \in \mathbb{G}^2$ and a scalar a. For the sake of simplicity, we assume that the adversary picks $(u_0, u_1) \in \mathbb{G}^2$ uniformly at random and picks a among the scalars of bit-length $\log(p)$. The challenger picks uniformly at random a bit $b \in \{0, 1\}$ and sets $u = u_b$. The client runs the delegation protocol with inputs u and a and delegates one exponentiation to the adversary acting as the server. The adversary has to guess the bit b.

Let us denote $(t_1, \sigma(g^{t_1}))$, $(t_2, \sigma(g^{t_2}))$, $\ldots$, $(t_\ell, \sigma(g^{t_\ell}))$ the pairs obtained from the random power generator $\mathcal{B}$ by the client $\mathcal{C}$. Since the random power generator $\mathcal{B}$ takes no inputs and outputs independent pairs, we can assume without loss of generality that the client $\mathcal{C}$ makes the ℓ queries to $\mathcal{B}$ in a first phase of the delegation protocol.

We denote $(\sigma(h), x)$ the unique pair group element/scalar queried by $\mathcal{C}$ to the server $\mathcal{S}$ (which is executed by the adversary $\mathcal{A}$ in an "honest-but-curious" way). Using generic group operations, $\mathcal{C}$ can only construct the corresponding group elements as:

$$h = u^{\gamma_1} \cdot g^{\gamma_2} (g^{t_1})^{\theta_1} \ldots (g^{t_\ell})^{\theta_\ell} \tag{15}$$

for some scalars $(\gamma_1, \gamma_2, \theta_1, \ldots, \theta_\ell) \in \mathbb{Z}_p^{\ell+2}$. We denote $h = u^{\gamma_1} g^r$ (with $r = \gamma_2 + t_1\theta_1 + \cdots + t_\ell\theta_\ell$) and $k = h^x$ the response of the server $\mathcal{S}$.

Eventually, the client $\mathcal{C}$ outputs[5] the encoding $\sigma(u^a)$ of the group element u^a and as above using generic group operations, it can only construct it as

$$u^a = u^{\alpha_1} g^{\alpha_2} k^{\alpha_3} \cdot g^{t_1\kappa_1} \cdots g^{t_\ell\kappa_\ell} \tag{16}$$

for some scalars $(\alpha_1, \alpha_2, \alpha_3, \kappa_1, \ldots, \kappa_\ell) \in \mathbb{Z}_p^{\ell+3}$.

As in the previous proofs, since $q = o(\sqrt{p})$, the client $\mathcal{C}$ is not able to compute the discrete logarithm of u in base g. This means that necessarily the exponents of g and u in Equation (16) cancel out. Recall that $k = h^x$, h being constructed as

[5] We do not assume that the adversary learns this value but only that the client $\mathcal{C}$ has to output it by the correctness property.

in Equation (15). Thus, taking only the discrete logarithms of powers of u in base u of this equation, we obtain

$$a = \alpha_1 + \alpha_3\gamma_1 x \mod p. \tag{17}$$

Similarly, taking only the discrete logarithm of powers of g of this equation, we obtain

$$0 = \alpha_2 + rx\alpha_3 + \kappa_1 t_1 + \cdots + \kappa_\ell t_\ell \mod p. \tag{18}$$

We denote τ_1 the number of group operations performed by $\mathcal{C}$ in the computation of h described in Equation (15) and τ_2 the number of group operations performed by $\mathcal{C}$ in the computation of u^a described in Equation (16).

By assumption, $\tau_1 + \tau_2 \leq c \log p + O(1)$. Furthermore, since $\mathcal{C}$ only used generic group operations, we have $\gamma_1 \leq 2^{\tau_1}$, $\alpha_i \leq 2^{\tau_1+\tau_2}$ for $i \in \{1, 2\}$ and $\alpha_3 \leq 2^{\tau_2}$.

The delegation protocol must ensure the privacy of u therefore in Equation (15), the value r such that the group element g^r masks u^{γ_1} must be different from 0. Otherwise, the adversary can simply try to find the (small) discrete logarithm of h in base u_0 or u_1 using for instance Shanks "baby steps, giant steps" or Pollard λ algorithm in time $O(\sqrt{\gamma_1}) = O(p^{c/2+o(1)})$. Combining Equations (17) and (18), we have:

$$\begin{aligned} ra &= r\alpha_1 + r\alpha_3\gamma_1 x - \gamma_1(\alpha_2 + rx\alpha_3 + \kappa_1 t_1 + \cdots + \kappa_\ell t_\ell) \mod p \\ &= r\alpha_1 - \gamma_1(\alpha_2 + \kappa_1 t_1 + \cdots + \kappa_\ell t_\ell) \mod p. \end{aligned}$$

with $r \neq 0$. Therefore, since $r = \gamma_2 + t_1\theta_1 + \cdots + t_\ell\theta_\ell$, the random scalar a can be written as:

$$a = \frac{r\alpha_1 - \gamma_1(\alpha_2 + \kappa_1 t_1 + \cdots + \kappa_\ell t_\ell)}{\gamma_2 + t_1\theta_1 + \cdots + t_\ell\theta_\ell} \mod p$$

and

$$a = \frac{\gamma_2\alpha_1 - \gamma_1\alpha_2 + \sum_{i=1}^{\ell} t_i(\theta_i\alpha_1 - \kappa_i\gamma_1)}{\gamma_2 + t_1\theta_1 + \cdots + t_\ell\theta_\ell} \mod p. \tag{19}$$

For fixed values $t_1, \ldots, t_\ell$, the number of scalars a that can be written in this form is upper-bounded by the product of number of α_1, α_2, γ_1, γ_2, θ_i's and κ_i's. We have, by Lemma 1

$$\alpha_1 \leq 2^{\tau_1+\tau_2} \quad \alpha_2 \leq 2^{\tau_1+\tau_2} \quad \gamma_1 \leq 2^{\tau_1} \quad \gamma_2 \leq 2^{\tau_1} \quad \theta_i \leq 2^{\tau_1} \quad \kappa_i \leq 2^{\tau_1+\tau_2}$$

for $i \in \{1, \ldots, \ell\}$. Therefore, the number of scalars a that can be written as in Equation (19) is upper-bounded by

$$2^{\tau_1+\tau_2} \times 2^{\tau_1+\tau_2} \times 2^{\tau_1} \times 2^{\tau_1} \times (2^{\tau_1})^\ell \times (2^{\tau_1+\tau_2})^\ell \leq (2^{\tau_1+\tau_2})^{2\ell+4}.$$

Since $2^{\tau_1+\tau_2} \leq p^{c+o(1)}$ with $c < 1/(2\ell + 4)$, we have shown that all scalars $a \in \mathbb{Z}_p$ cannot be written as in Equation (19) and therefore, the delegation protocol is not correct.

Remark 8 *It is worth noting that in the previous proof, we use only the fact that the scalar r (used in the exponent of the masking group element g^r) is not zero. It might be possible to improve our lower bound by using the much stronger privacy notion.*

7 Complexity Lower Bound for Multi-Round Protocols

7.1 Complexity Lower Bound for Two-Round Protocols

We consider the delegation of the exponentiation u^a with variable and public base u and secret exponent a. One can easily adapt the proof of Theorem 2 to the case where the client is allowed to delegate two group exponentiations in an adaptive way (i.e., in two communication rounds). Informally, Theorem 5 asserts that in this case the client needs to perform at least $\log(p)/4$ group operations (even if it is allowed to make an arbitrary constant number of queries to a random power generator for a generator $g \neq u$).

Theorem 5 *Let* $\mathsf{GroupGen}$ *be a group generator and let* $(\mathcal{C}, \mathcal{S})$ *be one client-server protocol for the delegated computation of the exponentiation* u^a *for the corresponding computation code* $\boldsymbol{\beta} = 101v$*. We assume that the client* $\mathcal{C}$ *is a generic group algorithm that uses*

- $c \log(p) + O(1)$ *generic group operations (for all groups* $\mathbb{G}$ *of primer order* p *output by* $\mathsf{GroupGen}(\lambda)$*) for some constant* c*,*
- $\ell = O(1)$ *queries to the (private) random power generator* $\mathcal{B}$
- *and* 2 *adaptive delegated exponentiation to the server* $\mathcal{S}$

If $c < 1/4$*, then* $(\mathcal{C}, \mathcal{S})$ *is not indistinguishable: there exists an honest-but-curious algorithm running in polynomial-time and a constant* $\kappa > 0$ *such that*

$$\Pr[\nu = 1 | \nu \leftarrow \mathbf{Exp}_{ind}(\mathcal{A}, \boldsymbol{\beta}, \lambda)] \geq 1 - \lambda^{-\kappa}.$$

Proof The proof is similar to the proof of Theorem 2.

We assume that the client $\mathcal{C}$ gets as input two encodings $\sigma(u)$, $\sigma(g)$ of *two* group elements u and g picked uniformly at random in $\mathbb{G}$ and one scalar a picked uniformly at random in $\mathbb{Z}_p$ and outputs the encoding $\sigma(u^a)$ of the group element u^a by making only

- q queries to the group oracle $\mathcal{G}$;
- ℓ queries to the (private) random power generator $\mathcal{B}$;
- 2 (adaptive) queries to the server oracle $\mathcal{S}$.

We assume that $q = c \log p + O(1)$ with $c < 1/4$. and we prove that it is not possible for $\mathcal{C}$ to compute $\sigma(u^a)$ in such a way that the server $\mathcal{S}$ learns no information on a. More precisely, the challenger picks uniformly at random a scalar $a^* \in \mathbb{Z}_p$ and a random bit b and sets $(a_b, a_{1-b}) = (a, a^*)$ (*i.e.* $\{a_0, a_1\} = \{a, a^*\}$ in a random order). The adversary (with the knowledge of the server $\mathcal{S}$'s transcript) has to guess the bit b.

Let us denote $(t_1, \sigma(g^{t_1}))$, $(t_2, \sigma(g^{t_2}))$, $\ldots$, $(t_\ell, \sigma(g^{t_\ell}))$ the pairs obtained from the random power generator $\mathcal{B}$ by the client $\mathcal{C}$. Since the random power generator $\mathcal{B}$ takes no inputs and outputs independent pairs, we can assume without loss of generality that the client $\mathcal{C}$ makes the ℓ queries to $\mathcal{B}$ in a first phase of the delegation protocol.

We denote $(\sigma(h_1), x_1)$ the first pair group element/scalar made by $\mathcal{C}$ to the server $\mathcal{S}$. Using generic group operations, $\mathcal{C}$ can only construct the corresponding group elements such that:

$$h = u^{\alpha'} \cdot g^{\kappa'} \cdot g^{t_1 \gamma'_1} \cdots g^{t_\ell \gamma'_\ell} \tag{20}$$

for some scalars $(\alpha', \kappa', \gamma'_1, \ldots, \gamma'_\ell)$. We denote $k_1 = h_1^{x_1}$ the response of $\mathcal{S}$.

We denote $(\sigma(h_2), x_2)$ the second pair group element/scalar made by $\mathcal{C}$ to the server $\mathcal{S}$. Using generic group operations, $\mathcal{C}$ can only construct the corresponding group elements such that:

$$h_2 = u^{\alpha''} \cdot g^{\kappa''} \cdot g^{t_1\gamma''_1} \cdots g^{t_\ell\gamma''_\ell} k_1^{\delta''} \tag{21}$$

for some scalars $(\alpha'', \kappa'', \gamma''_1, \ldots, \gamma''_\ell, \delta'', \varepsilon'')$. We denote $k_2 = h_2^{x_2}$ the response of the $\mathcal{S}$.

Eventually, the client $\mathcal{C}$ outputs the encoding $\sigma(u^a)$ of the group element u^a and as above using generic group operations, it can only construct it as

$$u^a = u^\alpha g^\kappa \cdot g^{t_1\gamma_1} \cdots g^{t_\ell\gamma_\ell} k_1^\delta k_2^\zeta \tag{22}$$

for some scalars $(\alpha, \kappa, \gamma_1, \ldots, \gamma_\ell, \delta, \varepsilon, \zeta, \eta)$. If we assume that $q = c \log n + O(1)$ (and in particular $q = o(\sqrt{p})$), the client $\mathcal{C}$ is not able to compute the discrete logarithm of u in base g. This means that necessarily the exponent of g in Equation (22) cancel out. Recall that $k_1 = h_1^{x_1}$ and $k_2 = h_2^{x_2}$, h_1 and h_2 being constructed as in Equation (20) and Equation (21). Thus, taking only the discrete logarithms of powers of u in base u of this equation, we obtain

$$a = \alpha + (\delta\alpha')x_1 + (\alpha''\zeta)x_2 + (\zeta\delta''\alpha')x_1x_2 \mod p. \tag{23}$$

For a random choice of $a \in \mathbb{Z}_p$, we have $a = \Omega(p)$. We denote τ_1 the number of group operations performed by $\mathcal{C}$ in the computation of h_1 described in Equation (20), τ_2 the number of group operations performed by $\mathcal{C}$ in the computation of h_2 described in Equation (21) and τ_3 the number of group operations performed by $\mathcal{C}$ in the computation of u^a described in Equation (22).

By assumption, $\tau_1 + \tau_2 + \tau_3 \leq c \log p + O(1)$. If we note $\rho_1 = \alpha$, $\rho_2 = \delta\alpha'$, $\rho_3 = \alpha''\zeta$ and $\rho_4 = \zeta\delta''\alpha'$ Equation (23) becomes

$$a = \rho_1 + \rho_2 x_1 + \rho_3 x_2 + \rho_4 x_1 x_2 \mod p$$

where x_1 and x_2 are known to the adversary. Furthermore, since $\mathcal{C}$ only used generic group operations, we have as above $\rho_i \leq p^{c+o(1)}$ for $i \in \{1, 2, 3, 4\}$.

The adversary can then try to decompose a_0 and a_1 as

$$a_i = \rho_{i,1} + \rho_{i,2} x_1 + \rho_{i,3} x_2 + \rho_{i,4} x_1 x_2 \mod p$$

with $\rho_{i,j} \leq p^{c+o(1)}$ for $i \in \{0, 1\}$ and $j \in \{1, 2, 3, 4\}$. For $a_b = a$, the algorithm from Section 3.3 will recover $\rho_{b,1}$, $\rho_{b,2}$, $\rho_{b,3}$ and $\rho_{b,4}$ in polynomial time. However, for a given pair (x_1, x_2) and a random $a_{1-b} = a^*$, there is only a negligible probability that such a decomposition exists (less than $(p^{c+o(1)})^4 = p^{4c+o(1)} = o(p)$ scalars can be written in this way). Thus, the adversary can simply run the Coppersmith-like algorithm on $(a_0, 1, x_1, x_2, x_1x_2)$ on one hand and on $(a_1, 1, x_1, x_2, x_1x_2)$ on the other hand and returns the bit b for which the algorithm returns a "small decomposition" on input $(a_b, 1, x_1, x_2, x_1x_2)$. By the previous analysis, its advantage is noticeable.

In this setting, the best delegation protocol (to our knowledge) requires $\log(p)/3$ group operations for the client: it is Protocol 5 from Section 5 (with $s = 2$) that do not take advantage of the fact that the second delegated exponentiation may depend on the first one.

If there exists a way to express the exponent a as a weighted sum

$$a = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_2 x_1 x_2 \mod p \tag{24}$$

with $\alpha_i \leq p^{1/4}$ for $i \in \{0, 1, 2\}$ for some arbitrary scalars x_1 and x_2 that do not reveal information on a, then the client may query the server the exponentiation $k_1 = u^{x_1}$ and subsequently $k_2 = (uk_1)^{x_2} = (u^{x_1+1})^{x_2}$ such that $u^a = u^{\alpha_0} k_1^{\alpha_1} k_2^{\alpha_2}$. Using Algorithm 1, this approach would make it possible for the client to compute u^a with roughly $\log(p)/4$ group operations by delegating two successive group exponentiations to the server (and in this case Theorem 5 will prove the optimality of this algorithm).

Even if the computational improvement from $\log(p)/3$ to $\log(p)/4$ group operations would be marginal in practice compared to the increase of the round complexity (and thus the latency of the protocol), it is an interesting theoretical open problem to study the existence of such decompositions (and to provide an efficient algorithm to construct them).

We can also consider the case where the client is allowed to delegate several group exponentiations in an adaptive way but in only two rounds. We obtain the following theorem:

Theorem 6 *Let* GroupGen *be a group generator and let* $(\mathcal{C}, \mathcal{S})$ *be one client-server protocol for the delegated computation of the exponentiation* u^a *for the corresponding computation code* $\boldsymbol{\beta} = 101v$*. We assume that the client* $\mathcal{C}$ *is a generic group algorithm that uses*

- $c \log(p) + O(1)$ *generic group operations (for all groups* $\mathbb{G}$ *of primer order* p *output by* $\mathsf{GroupGen}(\lambda)$*) for some constant* c*,*
- $\ell = O(1)$ *queries to the (private) random power generator* $\mathcal{B}$
- *and* s *simultaneous delegated exponentiation to the server* $\mathcal{S}$ *in two rounds*

If $c < 4/(4 + (s+1)^2)$*, then* $(\mathcal{C}, \mathcal{S})$ *is not indistinguishable: there exists an honest-but-curious algorithm running in polynomial-time and a constant* $\kappa > 0$ *such that*

$$\Pr[\nu = 1 | \nu \leftarrow \mathbf{Exp}_{ind}(\mathcal{A}, \boldsymbol{\beta}, \lambda)] \geq 1 - \lambda^{-\kappa}.$$

Proof The proof is similar to the proof of Theorem 3.

In particular, Theorem 6 asserts that in order to construct a delegation protocol in which the client performs only a constant number a group operations, then the round complexity of the protocol should be at least $\Omega(\sqrt{\log(p)})$. The proof of Theorem 6 actually shows the stronger result that even if the round complexity is $O(\sqrt{\log(p)})$, then the number of group operations for the client is also of order $\Omega(\sqrt{\log(p)})$ (and is therefore non-constant).

7.2 Complexity Lower Bound for Multiple-Round Protocols

For completeness, we mention that it is also possible to prove a lower bound on the efficiency of delegation protocols with any round complexity. For simplicity, we state only the complexity lower bound in the case of a delegation protocol that delegates the computation of s group exponentiations in s rounds (in an adaptive way). The lower bound is not as strong as the previous one since it decrease exponentially with s. Roughly speaking, Theorem 7 asserts that the best delegation protocol we can hope for requires $\Omega(\log\log(p))$ rounds in order to decrease the computational complexity of the client to only $O(\log\log(p))$ group operations.

Theorem 7 *Let* $\mathsf{GroupGen}$ *be a group generator and let* $(\mathcal{C},\mathcal{S})$ *be one client-server protocol for the delegated computation of the exponentiation* u^a *for the corresponding computation code* $\boldsymbol{\beta} = 101v$*. We assume that the client* $\mathcal{C}$ *is a generic group algorithm that uses*

- $c\log(p)+O(1)$ *generic group operations (for all groups* $\mathbb{G}$ *of primer order* p *output by* $\mathsf{GroupGen}(\lambda)$*) for some constant* c*,*
- $\ell = O(1)$ *queries to the (private) random power generator* $\mathcal{B}$
- *and* s *simultaneous delegated exponentiation to the server* $\mathcal{S}$

If $c < 2^{-s}$*, then* $(\mathcal{C},\mathcal{S})$ *is not indistinguishable: there exists an honest-but-curious algorithm running in polynomial-time and a constant* $\kappa > 0$ *such that*

$$\Pr[\nu = 1 | \nu \leftarrow \mathbf{Exp}_{ind}(\mathcal{A},\boldsymbol{\beta},\lambda)] \geq 1 - \lambda^{-\kappa}.$$

Proof The proof is again similar to the proof of Theorem 3.

8 Generic Constructions for Outsourcing Multi-Exponentiations

As mentioned in Section 4, even if one can fix Wang *et al.*'s protocol by using a larger Υ (such that the value χ is actually uniformly distributed over $\mathbb{Z}_p$), the resulting inefficient protocol would still not achieve the privacy security notion. Indeed, in their protocol the μ_1 and μ_3 used to mask the secret bases $u_{i,j}$ in Equation (2) and Equation (3) are always the same for all bases. In particular, an adversary against the privacy of this protocol can simply picked bases $(u^0_{1,1}, u^0_{1,2})$ and $(u^1_{1,1}, u^1_{1,2})$ in $\mathbf{Exp}_{ind}(\mathcal{A})$ such that $u^0_{1,1}/u^0_{1,2} \neq u^1_{1,1}/u^1_{1,2}$. Since, from Equation (3), we know that $w_{1,1}/w_{1,2} = u^b_{1,1}/u^b_{1,2}$ then it can determine the bit b used in the experiment with certainty.

We thus give in this section several protocols for outsourcing multi-exponentiations

$$(u_1,\ldots,u_n,a_1,\ldots,a_n) \mapsto {u_1}^{a_1}\cdots{u_n}^{a_n}.$$

Their security is stated in Theorem 8. The proof of this theorem is similar to the proof of Theorem 1 and is omitted.

Theorem 8 *Let* $\mathsf{GroupGen}$ *be a group generator, let* λ *be a security parameter and let* $\mathbb{G}$ *be a group of primer order* p *output by* $\mathsf{GroupGen}(\lambda)$*. Let* $(\mathcal{C},\mathcal{S})$ *be one client-server protocol for the delegated computation of the multi-exponentiation* ${u_1}^{a_1}\cdots{u_n}^{a_n}$ *described in Protocols 10 – 14 (for the corresponding computation code* $\boldsymbol{\beta}$ *given in their description). The protocol* $(\mathcal{C},\mathcal{S})$ *satisfies* $(\tau,0)$*-indistinguishability against a malicious adversary for any time* τ*.*

Protocol 10: $100f$ (and $101f$)

Input: $u_1, \ldots, u_n \in \mathbb{G}$, $a_1, \ldots, a_n \in \mathbb{Z}_p$
Output: $u_1^{a_1} \cdots u_n^{a_n} \in \mathbb{G}$
  for i from 1 to n **do**
    $(u_i^{k_i}, k_i) \leftarrow \mathcal{B}(i)$
    $h_i \leftarrow \mathcal{S}(u_i, a_i - k_i \bmod p)$
  end for
  return $\prod_{i=1}^{n} h_i u_i^{k_i}$

Fig. 5 Delegation protocols for fixed base multi-exponentiation

8.1 Construction for Outsourcing Fixed Based Multi-Exponentiation

When the bases $(u_1, \ldots, u_n)$ are fixed, one can assume that $\mathcal{C}$ can use a random power generator $\mathcal{B}(i)$ for each u_i. As for the single exponentiation case, the cases $111f$, $110f$ and $011f$ are trivial or do not make sense.

We give Protocol 10 in case $100f$ where the bases are public, the exponents private and the result private. This protocol obviously work in the cases where the exponents or the result become public (case $101f$), but could probably be improved in these latter cases.

This protocol does not apply when the bases are private and exponents public (case $010f$), but one can instead use Protocol 11.

8.2 Construction for Outsourcing Variable Base Multi-Exponentiation

Since multi-exponentiations are at least as difficult as single exponentiations, lower bounds obtained in Section 6 show that it is impossible to construct a protocol using a constant number of operations in $\mathbb{G}$ when something is secret and the bases are variable. This gives further evidence that the protocols given in [45] cannot be private.

When the bases $(u_1, \ldots, u_n)$ are variable, one cannot assume that $\mathcal{C}$ can use a random power generator $\mathcal{B}(i)$ for each u_i, but he can still use one for the generator g, that we denote $\mathcal{B}$ in the following constructions.

As for the single exponentiation case, the cases $111v$, $110v$ and $011v$ are trivial or do not make sense.

We give Protocol 11 in case $011v$ where the bases are private, the exponents public and the result public and Protocol 12 in case $101v$ where the bases are public, the exponents private and the result public.

Finally, we give three protocols for the cases $100v$, $010v$ and $000v$ (Protocols 12, 13 and 14, respectively) which are basically parallel repetitions of the protocols for single exponentiation for the same cases. One may be tempted to reuse masks generated by the random power generator $\mathcal{B}$ for several private bases u_i (for $i \in \{1, \ldots, n\}$). However, one can prove using our techniques from Section 6 that this would result in insecure protocols.

9 Conclusion and Future Work

All our results on (one-round) secure delegation of group exponentation are collected in Table 2. In addition, we also provide protocols and lower-bounds for

Protocol 11: $011v$

Input: $u_1, \ldots, u_n \in \mathbb{G}$, $a_1, \ldots, a_n \in \mathbb{Z}_p$
Output: $u_1^{a_1} \cdots u_n^{a_n} \in \mathbb{G}$

```
for j from 1 to n do
    for i from 1 to s do
        g_i ←R G
    end for
    I ←R P_m({1,...,s})                    ▷ random subset of cardinal m of {1,...,s}
    g_{s+1} ← u_j · ∏_{i∈I} g_i
    for i from 1 to s+1 do
        h_i ← S(g_i, a_j)
    end for
    v_j ← h_{s+1} / ∏_{i∈I} h_i
end for
return v_1 ··· v_n
```

Protocol 12: $100v$ (and $101v$)

Input: $u_1, \ldots, u_n \in \mathbb{G}$, $a_1, \ldots, a_n \in \mathbb{Z}_p$
Output: $u_1^{a_1} \cdots u_n^{a_n} \in \mathbb{G}$

```
T ← ⌈p^{1/s+1}⌉
for j from 1 to n do
    for i from 1 to s do
        h_{i,j} ← S(u_j, T^i)
    end for
    temp ← a_j
    for i from s down to 0 do
        a_{i,j} ← temp div T^i             ▷ a_j = a_{s,j}·T^s + ··· + a_{1,j}T + a_{0,j}
        temp ← temp − a_{i,j}·T^i
    end for
end for
return ∏_{j=1}^n u_j^{a_0} ∏_{i=1}^s h_{i,j}^{a_{i,j}}        ▷ using Algorithm 1
```

Protocol 13: $010v$

Input: $u_1, \ldots, u_n \in \mathbb{G}$, $a_1, \ldots, a_n \in \mathbb{Z}_p$
Output: $u_1^{a_1} \cdots u_n^{a_n} \in \mathbb{G}$

```
for i from 1 to n do
    (g^{r_i}, r_i) ← B(·)
    h_{1,i} ← S(u_i · g^{r_i}, a_i)
end for
(g^s, s) ← B(·)
(g^t, t) ← B(·)
k ← (t − ∑_{i=1}^s r_i a_i)/s mod p
h_2 ← S(g^s, k)
return ∏_{i=1}^s h_{1,i} h_2 g^t
```

Protocol 14: $000v$ (and $001v$)

Input: $u_1, \ldots, u_n \in \mathbb{G}$, $a_1, \ldots, a_n \in \mathbb{Z}_p$
Output: $u_1^{a_1} \cdots u_n^{a_n} \in \mathbb{G}$

```
for j from 1 to n do
    (g^{k_{1,j}}, k_{1,j}) ← B(·)
    v_j ← u_j · g^{k_{1,j}}
    h_{1,j} ← v_j^{a_j}        ▷ delegated using Protocol 12; h_{1,j} = v_j^{a_j} = u_j^{a_j} · g^{a_j k_{1,j}}
    (g^{k_{2,j}}, k_{2,j}) ← B(·)
    h_{2,j} ← S(g, −a_j k_{1,j} − k_{2,j} mod p)        ▷ h_{2,j} = g^{−a_j k_{1,j} − k_{2,j}}
end for
return ∏_{j=1}^n h_{1,j} · h_{2,j} · g^{k_{2,j}}
```

Fig. 6 Delegation protocols for variable base multi-exponentiation

multi-exponentiations and lower bounds for multi-round delegation of exponentiation protocols. As a future work, understanding the relationship between computational efficiency and memory usage is vital when implementing delegation protocols. In particular, it is interesting to propose efficient delegation protocols and to improve our lower bounds in settings where the memory complexity of the client is limited. It is also interesting to provide provably secure protocols and complexity lower bounds for exponentiation protocols in groups of *unknown* order (which are of interest to delegate the computation of an RSA signature) [17, 32].

Acknowledgments

The authors are supported in part by the French ANR ALAMBIC (ANR-16-CE39-0006). The authors thank Guillaume Hanrot and Damien Stehlé for helpful discussions, and Olivier Billet for his comments and for pointing out references.

References

1. Martin Albrecht, Shi Bai, David Cadé, Xavier Pujol, and Damien Stehlé. fplll-4.0, a floating-point LLL implementation. `http://perso.ens-lyon.fr/damien.stehle`.
2. Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Xiaodong Song. Provable data possession at untrusted stores. In Ning et al. [37], pages 598–609.
3. Roberto Maria Avanzi. The complexity of certain multi-exponentiation techniques in cryptography. *Journal of Cryptology*, 18(4):357–373, September 2005.
4. László Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.
5. Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 1–16, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.
6. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532, Gold Coast, Australia, December 9–13, 2001. Springer, Heidelberg, Germany.
7. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.
8. Victor Boyko, Marcus Peinado, and Ramarathnam Venkatesan. Speeding up discrete log and factoring based schemes via precomputations. In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 221–235, Espoo, Finland, May 31 – June 4, 1998. Springer, Heidelberg, Germany.
9. Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David Bruce Wilson. Fast exponentiation with precomputation (extended abstract). In Rainer A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT'92*, volume 658 of *Lecture Notes in Computer Science*, pages 200–207, Balatonfüred, Hungary, May 24–28, 1993. Springer, Heidelberg, Germany.
10. Sébastien Canard, Julien Devigne, and Olivier Sanders. Delegating a pairing can be both secure and efficient. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS 14: 12th International Conference on Applied Cryptography and Network Security*, volume 8479 of *Lecture Notes in Computer Science*, pages 549–565, Lausanne, Switzerland, June 10–13, 2014. Springer, Heidelberg, Germany.
11. Bren Cavallo, Giovanni Di Crescenzo, Delaram Kahrobaei, and Vladimir Shpilrain. Efficient and secure delegation of group exponentiation to a single server. In *Radio Frequency Identification. Security and Privacy Issues - 11th International Workshop, RFIDsec 2015*, LNCS, pages 156–173, 2015.

12. Xiaofeng Chen, Jin Li, Jianfeng Ma, Qiang Tang, and Wenjing Lou. New algorithms for secure outsourcing of modular exponentiations. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *ESORICS 2012: 17th European Symposium on Research in Computer Security*, volume 7459 of *Lecture Notes in Computer Science*, pages 541–556, Pisa, Italy, September 10–12, 2012. Springer, Heidelberg, Germany.
13. Céline Chevalier, Fabien Laguillaumie, and Damien Vergnaud. Privately outsourcing exponentiation to a single server: Cryptanalysis and optimal constructions. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016: 21st European Symposium on Research in Computer Security, Part I*, volume 9878 of *Lecture Notes in Computer Science*, pages 261–278, Heraklion, Greece, September 26–30, 2016. Springer, Heidelberg, Germany.
14. Benoît Chevallier-Mames, Jean-Sébastien Coron, Noel McCullagh, David Naccache, and Michael Scott. Secure delegation of elliptic-curve pairing. In *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010*, volume 6035 of *LNCS*, pages 24–35. Springer, 2010.
15. Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In Ueli M. Maurer, editor, *Advances in Cryptology – EUROCRYPT'96*, volume 1070 of *Lecture Notes in Computer Science*, pages 178–189, Saragossa, Spain, May 12–16, 1996. Springer, Heidelberg, Germany.
16. Don Coppersmith. Finding a small root of a univariate modular equation. In Ueli M. Maurer, editor, *Advances in Cryptology – EUROCRYPT'96*, volume 1070 of *Lecture Notes in Computer Science*, pages 155–165, Saragossa, Spain, May 12–16, 1996. Springer, Heidelberg, Germany.
17. Giovanni Di Crescenzo, Matluba Khodjaeva, Delaram Kahrobaei, and Vladimir Shpilrain. Secure delegation to a single malicious server: Exponentiation in rsa-type groups. In *7th IEEE Conference on Communications and Network Security, CNS 2019, Washington, DC, USA, June 10-12, 2019*, pages 1–9. IEEE, 2019.
18. Peter de Rooij. Efficient exponentiation using precomputation and vector addition chains. In Alfredo De Santis, editor, *Advances in Cryptology – EUROCRYPT'94*, volume 950 of *Lecture Notes in Computer Science*, pages 389–399, Perugia, Italy, May 9–12, 1995. Springer, Heidelberg, Germany.
19. Peter de Rooij. On Schnorr's preprocessing for digital signature schemes. *Journal of Cryptology*, 10(1):1–16, December 1997.
20. Yong Ding, Zheng Xu, Jun Ye, and Kim-Kwang Raymond Choo. Secure outsourcing of modular exponentiations under single untrusted programme model. *J. Comput. Syst. Sci.*, 90:1–13, 2017.
21. Steven D. Galbraith. *Mathematics of public key cryptography*. Cambridge University Press, 2012.
22. Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.
23. Aurore Guillevic and Damien Vergnaud. Algorithms for outsourcing pairing computation. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 193–211. Springer, 2014.
24. Mathias Herrmann. *Lattice-based Cryptanalysis using Unravelled Linearization*. PhD thesis, Ruhr-Universität Bochum, 2011.
25. Susan Hohenberger and Anna Lysyanskaya. How to securely outsource cryptographic computations. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 264–282, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany.
26. Nick Howgrave-Graham. Finding small roots of univariate modular equations revisited. In Michael Darnell, editor, *6th IMA International Conference on Cryptography and Coding*, volume 1355 of *Lecture Notes in Computer Science*, pages 131–142, Cirencester, UK, December 17–19, 1997. Springer, Heidelberg, Germany.
27. Ari Juels and Burton S. Kaliski Jr. Pors: proofs of retrievability for large files. In Ning et al. [37], pages 584–597.
28. Charanjit S. Jutla. On finding small solutions of modular multivariate polynomial equations. In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT'98*, volume 1403

of *Lecture Notes in Computer Science*, pages 158–170, Espoo, Finland, May 31 – June 4, 1998. Springer, Heidelberg, Germany.
29. Mehmet Sabır Kiraz and Osmanbey Uzunkol. Efficient and verifiable algorithms for secure outsourcing of cryptographic computations. *International Journal of Information Security*, pages 1–19, 2015.
30. Arjen K. Lenstra, Hendrik W. Jr. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.
31. Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In Yvo Desmedt, editor, *Advances in Cryptology – CRYPTO'94*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107, Santa Barbara, CA, USA, August 21–25, 1994. Springer, Heidelberg, Germany.
32. Thierry Mefenza and Damien Vergnaud. Cryptanalysis of server-aided RSA protocols with private-key splitting. *Comput. J.*, 62(8):1194–1213, 2019.
33. Silvio Micali, Rafael Pass, and Alon Rosen. Input-indistinguishable computation. In *47th Annual Symposium on Foundations of Computer Science*, pages 367–378, Berkeley, CA, USA, October 21–24, 2006. IEEE Computer Society Press.
34. Bodo Möller. Algorithms for multi-exponentiation. In Serge Vaudenay and Amr M. Youssef, editors, *SAC 2001: 8th Annual International Workshop on Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 165–180, Toronto, Ontario, Canada, August 16–17, 2001. Springer, Heidelberg, Germany.
35. Phong Q. Nguyen, Igor E. Shparlinski, and Jacques Stern. Distribution of modular sums and the security of server aided exponentiation. In *Workshop on Comp. Number Theory and Crypt*, pages 1–16, 1999.
36. Phong Q. Nguyen and Damien Stehlé. Floating-point LLL revisited. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 215–233, Aarhus, Denmark, May 22–26, 2005. Springer, Heidelberg, Germany.
37. Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors. *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. ACM, 2007.
38. *Sage Mathematics Software*, 2012. `http://www.sagemath.org`.
39. Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991.
40. Hovav Shacham and Brent Waters. Compact proofs of retrievability. In Josef Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 90–107, Melbourne, Australia, December 7–11, 2008. Springer, Heidelberg, Germany.
41. Mehul A. Shah, Mary Baker, Jeffrey C. Mogul, and Ram Swaminathan. Auditing to keep online storage services honest. In Galen C. Hunt, editor, *Proceedings of HotOS'07: 11th Workshop on Hot Topics in Operating Systems, May 7-9, 2005, San Diego, California, USA*. USENIX Association, 2007.
42. Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266, Konstanz, Germany, May 11–15, 1997. Springer, Heidelberg, Germany.
43. Benjamin Smith. Easy scalar decompositions for efficient scalar multiplication on elliptic curves and genus 2 Jacobians. *Contemporary Mathematics Series*, 637:15, 2015.
44. Ernst Gabor Straus. Problems and solutions: Addition chains of vectors. American Mathematical Monthly 71, 1964. p. 806–808.
45. Yujue Wang, Qianhong Wu, Duncan S. Wong, Bo Qin, Sherman S. M. Chow, Zhen Liu, and Xiao Tan. Securely outsourcing exponentiations with single untrusted program for cloud storage. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *ESORICS 2014: 19th European Symposium on Research in Computer Security, Part I*, volume 8712 of *Lecture Notes in Computer Science*, pages 326–343, Wroclaw, Poland, September 7–11, 2014. Springer, Heidelberg, Germany.
46. Ikkwon Yie. Cryptanalysis of Elgamal type digital signature schemes using integer decomposition. *Trends in Mathematics*, 8(1):167–175, June 2005.

A Practical Example of the Attack on Wang et al.'s algorithm

In this section, we provide a concrete example of the attack described in Section 4.

For p a 256-bit prime, recovering roots which have less than 110 bits implies the reduction of lattice of a dimension 3 with entries of largest bit-size around 360. Indeed, as mentioned in Section 4, we construct the matrix

$$M = \begin{pmatrix} p & 0 & 0 \\ 0 & pY & 0 \\ c & bY & X \end{pmatrix}$$

whose rows are formed by the coefficients of the polynomials p, pyY and $P(xX, yY)$ in the basis $(1, X, Y)$.

Let us consider a 256-bit prime

$$p = \texttt{10A98A92 2EC19799 E81125D6 D3B0C1EB}$$
$$\texttt{0D6FE6D8 67D32C56 492FC5521 D1398C33}.$$

The bound X and Y are set as

$$X = Y = \texttt{42E0EA80 D850A1EF BDAFD0E7 6115D4}.$$

The Coppersmith matrix M constructed as previously mentioned and the corresponding LLL-reduced matrix M_{red} are given in Figure 7.

$$M = \begin{pmatrix} \texttt{10A98A92 2EC19799 E81125D6 D3B0C1EB} & 0 & 0 \\ \texttt{0D6FE6D8 67D32C56 492FC5521 D1398C33} & & \\ 0 & \texttt{45A59564 B7A06BF8 9C6D4ED4 D08C2DC1} & \\ & \texttt{E2F88EB6 24EE3B81 02FA2AA7 26D36A9A} & 0 \\ & \texttt{2566E317 86B0B9A5 CD40EB20 7B493C} & \\ \texttt{E71E5733 9704C37D 34AB38DA 83EA7927} & \texttt{3B82CE9A 11B11310 D12A193C E17FE212} & \\ \texttt{D8E5676D 09D1923A 2143A36F B4F01EAE} & \texttt{B62B436F 73B01FF6 344FA7B5 F9D70DB5} & \texttt{42E0EA80 D850A1EF BDAFD0E7 6115D4} \\ & \texttt{5A89EE31 CCFD3555 126BA888 D138F4} & \end{pmatrix}$$

$$M_{\mathsf{red}} = \begin{pmatrix} \texttt{38F18511 39F7E280 204C22CC F39537F} & \texttt{−30D8F3AD 3FABCE99 D5AF4BC6 66B620FD} & \texttt{−1AE68949 B7191B94 7FE4E3CA 8DAE9011} \\ \texttt{A9ECD308 6EBE1D8D 72E1DC04 E57429} & \texttt{F9877F0C 3442C051 634A8E7B 64EA20} & \texttt{15D27BC9 ED826539 2AEEDBC4 EE7FBC} \\ \texttt{51B23E00 A5CFD8D9 BA767703 77C8BA2D} & \texttt{−21CF42F7 F543572A 10EA8711 75A197B7} & \texttt{−4CBC8A4D 547515B7 B2679846 C4584C6C} \\ \texttt{1370FA7E CB28EE35 4DA3E306 9A4877} & \texttt{B6DE5AA1 4089DEDD 5936F3E1 2D17F4} & \texttt{0824AEEC 452BDF56 CB042105 E38B34} \\ \texttt{87160B64 1EA1A7B0 CF27D073 400D2950} & \texttt{47D1FC34 CC4468C7 876C2516 AD3069EA} & \texttt{706BD602 2D5F627C 3D5F0F0C 68E6A0E5} \\ \texttt{1F9D68CF 50BFBAE5 AF103485 7F15677D} & \texttt{0D122117 7A0C3DCE A8A4F500 2EEC24A0} & \texttt{E6F72D54 69080AC7 55CADFEF 1312F3D8} \end{pmatrix}$$

Fig. 7 Example of an attack against [45]

The two first vectors allow to recover

$$\begin{cases} x_0 = \texttt{37921F2A 890AA857 DAC77BBF 803B5D} \\ y_0 = \texttt{2379CD0E 21A56BC1 33CAA48C 43B4B2}. \end{cases}$$

in less than 0.1 second on a standard laptop, using Sage math software [38] and fplll [1] . These vectors are of norms of size respectively 245 and 256 bits.

Remark 9 *Note that an alternative attack can also be adapted from Yie's paper [46], which aimed at recovering an ElGamal signature secret key when two signatures with small message nonces are available. Indeed, during the verification of an ElGamal signature, an equation of the form $h(m) = xr + ks \mod q$ is verified, where x (the secret key) and k (the nonce) are unknown, and r and s are part of the signature of the message m. Two such equations make it possible to get rid of x, and we are back to an equation like Eq. 5. Yie adapted the algorithm of Gallant, Lambert and Vanstone* GLV-Dec. *This algorithm uses the extended Euclidean algorithm and has a complexity of $O(\log_2(q)^3)$.*

B Outsourcing Exponentiations in Groups with Efficient Inverses

Let GroupGen be a group generator which takes as input a security parameter λ. It provides a set *params* which contains a description of a (multiplicative) group $(\mathbb{G}, \cdot)$, the group order, say $p = |\mathbb{G}|$, and one generator g. In this Appendix, we consider a variant of the generic

group model in $\mathbb{G}$ where the computation of group inverse is easy. This generic group is still implemented by choosing a random encoding $\sigma : \mathbb{G} \longrightarrow \{0,1\}^m$ (with $2^m > p$). As above, a generic algorithm $\mathcal{A}$ takes as input (in addition to the group order p) their image under σ. This way, all $\mathcal{A}$ can test is group elements equality (by encoding equality). $\mathcal{A}$ is also given access to an oracle $\mathcal{G}$ computing group multiplication: taking $\sigma(g_1)$ and $\sigma(g_2)$ encodings of two group elements $g_1, g_2 \in \mathbb{G}$ and a sign in $s \in \{-1, +1\}$ as inputs and returning $\sigma(g_1 \cdot g_2^s)$ the encoding of the product $g_1 \cdot g_2^s \in G$ (i.e., $g_1 \cdot g_2$ or g_1/g_2) . We assume again that $\mathcal{A}$ submits to the oracle only encodings of elements it had previously received. In this enhanced generic group model, we have the following lemma analogous to Lemma 1:

Lemma 4 *Considering this enhanced generic group model, let* GroupGen *be a group generator, let $\mathbb{G}$ be a group of prime order p output by* GroupGen *and let $\mathcal{A}$ be a generic algorithm in $\mathbb{G}$. If $\mathcal{A}$ is given as inputs encodings $\sigma(g_1),\ldots,\sigma(g_n)$ of group elements $g_1, \ldots, g_n \in \mathbb{G}$ (for $n \in \mathbb{N}$) and outputs the encoding $\sigma(h)$ of a group element $h \in \mathbb{G}$ in time τ, then there exists integers $\alpha_1, \ldots, \alpha_n \in \mathbb{Z}$ such that $h = g_1^{\alpha_1} \ldots g_n^{\alpha_n}$ and $\max(|\alpha_1|, \ldots, |\alpha_n|) \leq 2^\tau$.*

Proof We can – as in the proof of Lemma 1 – define a map $\pi : \{0,1\}^m \to \mathbb{Z}^n$ which associates to each encoding obtained by $\mathcal{A}$ during its execution an n-dimensional vector in $\mathbb{Z}^n$. For each input encoding $\sigma(g_i)$, $\pi(\sigma(g_i))$ is defined as the i-th vector from the $\mathbb{Z}^n$ canonical basis (for $i \in \{1, \ldots, n\}$) and for each encoding $\sigma(h_1)$ and $\sigma(h_2)$ and each sign $s \in \{-1, 1\}$ queried to $\mathcal{G}$, $\pi(\sigma(h_1 \cdot h_2)^s) = \pi(\sigma(h_1)) + s \cdot \pi(\sigma(h_2)$. By construction, during the whole execution of $\mathcal{A}$, we have $\pi(\sigma(h)) = (\alpha_1, \ldots, \alpha_n)$ if and only if $h = g_1^{\alpha_1} \ldots g_n^{\alpha_n}$ for all encodings $\sigma(h)$. As in the proof of Lemma 1, the ℓ_∞-norm of $\pi(\sigma(h_1 \cdot h_2^s))$ is upper-bounded by $\ell_\infty(\pi(\sigma(h_1))) + \ell_\infty(\pi(\sigma(h_2))$ Since the ℓ_∞-norm of the input encodings $\pi(\sigma(g_i))$ is equal to 1 (for $i \in \{1, \ldots, n\}$) and the ℓ_∞-norm of encodings at most doubles for each query to $\mathcal{G}$, we obtained the claimed result.

In this setting, we can consider a variant of Algorithm 1 that computes the multi-exponentiation $\prod_{i=1}^t g_i^{x_i}$, for $g_1, \ldots, g_t \in \mathbb{G}$ and $x_1, \ldots, x_t \in \mathbb{N}$ by interleaving signed expansions of exponents. In particular, we can use the *w-ary non-adjacent form* method which guarantees that on average there will be fewer group multiplications for the same window size w (see [3,34] for details). In this case, the precomputation stage generates $\prod_{1 \leq i \leq t} g_i^{E_i}$ (and the algorithm can use the values $\prod_{1 \leq i \leq t} g_i^{-E_i}$) for all non-zero t-tuples $(E_1, \ldots, E_t) \in \{0, \ldots, 2^w - 1\}^t$ at no extra storage-cost). The total cost is thus for the precomputation phase $t2^{tw-2} - t$ multiplications and t squarings and overall less than $\ell/(w + 1/t) \leq \ell/w$ multiplications on average and ℓ squarings. For $t = 2$, the cost is again minimal for w around $1/2 \log \ell - \log \log \ell$ with $\ell(1 + 3/\log \ell) = \ell(1 + o(1))$ multiplications overall. Therefore, the method does not improve the asymptotic complexity (at least when the precomputation stage and the storage are not strongly limited). We can replace the use of Algorithm 1 by this variant but this does not improve the asymptotic complexity of the delegation protocols in the number of generic group operations (even with efficient inverses).

Actually, this fact is not surprising, since we can replace the use of Lemma 1 in the proof of our lower bound complexities (Theorems 2 – 7) by the use of Lemma 4 to obtain the same lower bounds for delegation protocols in the enhanced generic group model with inverses.